

FingerTrees

Benedikt Nordhoff Stefan Körner Peter Lammich

August 7, 2022

Abstract

We implement and prove correct 2-3 finger trees. Finger trees are a general purpose data structure, that can be used to efficiently implement other data structures, such as priority queues. Intuitively, a finger tree is an annotated sequence, where the annotations are elements of a monoid. Apart from operations to access the ends of the sequence, the main operation is to split the sequence at the point where a *monotone predicate* over the sum of the left part of the sequence becomes true for the first time. The implementation follows the paper of Hinze and Paterson[1]. The code generator can be used to get efficient, verified code.

Contents

1	2-3 Finger Trees	2
1.1	Datatype definition	2
1.1.1	Invariant	3
1.1.2	Abstraction to Lists	4
1.2	Operations	5
1.2.1	Empty tree	5
1.2.2	Annotation	5
1.2.3	Appending	6
1.2.4	Convert list to tree	8
1.2.5	Detaching leftmost/rightmost element	8
1.2.6	Concatenation	18
1.2.7	Splitting	23
1.2.8	Folding	41
1.2.9	Number of elements	42
1.3	Hiding the invariant	43
1.3.1	Datatype	43
1.3.2	Definition of Operations	45
1.3.3	Correctness statements	47
1.4	Interface Documentation	52

1 2-3 Finger Trees

```
theory FingerTree
imports Main
begin
```

We implement and prove correct 2-3 finger trees as described by Ralf Hinze and Ross Paterson[1].

This theory is organized as follows: Section 1.1 contains the finger-tree datatype, its invariant and its abstraction function to lists. The Section 1.2 contains the operations on finger trees and their correctness lemmas. Section 1.3 contains a finger tree datatype with implicit invariant, and, finally, Section 1.4 contains a documentation of the implemented operations.

Technical Issues As Isabelle lacks proper support of namespaces, we try to simulate namespaces by locales.

The problem is, that we define lots of internal functions that should not be exposed to the user at all. Moreover, we define some functions with names equal to names from Isabelle’s standard library. These names make perfect sense in the context of FingerTrees, however, they shall not be exposed to anyone using this theory indirectly, hiding the standard library names there. Our approach puts all functions and lemmas inside the locale *FingerTree_loc*, and then interprets this locale with the prefix *FingerTree*. This makes all definitions visible outside the locale, with qualified names. Inside the locale, however, one can use unqualified names.

1.1 Datatype definition

```
locale FingerTreeStruc-loc
```

Nodes: Non empty 2-3 trees, with all elements stored within the leafs plus a cached annotation

```
datatype ('e,'a) Node = Tip 'e 'a |
  Node2 'a ('e,'a) Node ('e,'a) Node |
  Node3 'a ('e,'a) Node ('e,'a) Node ('e,'a) Node
```

Digit: one to four ordered Nodes

```
datatype ('e,'a) Digit = One ('e,'a) Node |
  Two ('e,'a) Node ('e,'a) Node |
  Three ('e,'a) Node ('e,'a) Node ('e,'a) Node |
  Four ('e,'a) Node ('e,'a) Node ('e,'a) Node ('e,'a) Node
```

FingerTreeStruc: The empty tree, a single node or some nodes and a deeper tree

```
datatype ('e, 'a) FingerTreeStruc =
  Empty |
  Single ('e, 'a) Node |
  Deep 'a ('e, 'a) Digit ('e, 'a) FingerTreeStruc ('e, 'a) Digit
```

1.1.1 Invariant

```
context FingerTreeStruc-loc
begin
```

Auxiliary functions

Readout the cached annotation of a node

```
primrec gmn :: ('e, 'a::monoid-add) Node  $\Rightarrow$  'a where
  gmn (Tip e a) = a |
  gmn (Node2 a -) = a |
  gmn (Node3 a - -) = a
```

The annotation of a digit is computed on the fly

```
primrec gmd :: ('e, 'a::monoid-add) Digit  $\Rightarrow$  'a where
  gmd (One a) = gmn a |
  gmd (Two a b) = (gmn a) + (gmn b) |
  gmd (Three a b c) = (gmn a) + (gmn b) + (gmn c) |
  gmd (Four a b c d) = (gmn a) + (gmn b) + (gmn c) + (gmn d)
```

Readout the cached annotation of a finger tree

```
primrec gmft :: ('e, 'a::monoid-add) FingerTreeStruc  $\Rightarrow$  'a where
  gmft Empty = 0 |
  gmft (Single nd) = gmn nd |
  gmft (Deep a - -) = a
```

Depth and cached annotations have to be correct

```
fun is-leveln-node :: nat  $\Rightarrow$  ('e, 'a) Node  $\Rightarrow$  bool where
  is-leveln-node 0 (Tip -)  $\longleftrightarrow$  True |
  is-leveln-node (Suc n) (Node2 - n1 n2)  $\longleftrightarrow$ 
    is-leveln-node n n1  $\wedge$  is-leveln-node n n2 |
  is-leveln-node (Suc n) (Node3 - n1 n2 n3)  $\longleftrightarrow$ 
    is-leveln-node n n1  $\wedge$  is-leveln-node n n2  $\wedge$  is-leveln-node n n3 |
  is-leveln-node -  $\longleftrightarrow$  False
```

```
primrec is-leveln-digit :: nat  $\Rightarrow$  ('e, 'a) Digit  $\Rightarrow$  bool where
  is-leveln-digit n (One n1)  $\longleftrightarrow$  is-leveln-node n n1 |
  is-leveln-digit n (Two n1 n2)  $\longleftrightarrow$  is-leveln-node n n1  $\wedge$ 
    is-leveln-node n n2 |
  is-leveln-digit n (Three n1 n2 n3)  $\longleftrightarrow$  is-leveln-node n n1  $\wedge$ 
```

$is\text{-leveln}\text{-node } n \ n2 \wedge is\text{-leveln}\text{-node } n \ n3 \mid$
 $is\text{-leveln}\text{-digit } n \ (Four \ n1 \ n2 \ n3 \ n4) \longleftrightarrow is\text{-leveln}\text{-node } n \ n1 \wedge$
 $is\text{-leveln}\text{-node } n \ n2 \wedge is\text{-leveln}\text{-node } n \ n3 \wedge is\text{-leveln}\text{-node } n \ n4$

primrec $is\text{-leveln}\text{-ftree} :: nat \Rightarrow ('e, 'a) \text{FingerTreeStruc} \Rightarrow bool$ **where**
 $is\text{-leveln}\text{-ftree } n \ Empty \longleftrightarrow True \mid$
 $is\text{-leveln}\text{-ftree } n \ (Single \ nd) \longleftrightarrow is\text{-leveln}\text{-node } n \ nd \mid$
 $is\text{-leveln}\text{-ftree } n \ (Deep \ - \ l \ t \ r) \longleftrightarrow is\text{-leveln}\text{-digit } n \ l \wedge$
 $is\text{-leveln}\text{-digit } n \ r \wedge is\text{-leveln}\text{-ftree } (Suc \ n) \ t$

primrec $is\text{-measured}\text{-node} :: ('e, 'a :: monoid\text{-add}) \text{Node} \Rightarrow bool$ **where**
 $is\text{-measured}\text{-node } (Tip \ -) \longleftrightarrow True \mid$
 $is\text{-measured}\text{-node } (Node2 \ a \ n1 \ n2) \longleftrightarrow ((is\text{-measured}\text{-node } n1) \wedge$
 $(is\text{-measured}\text{-node } n2)) \wedge (a = (gmn \ n1) + (gmn \ n2)) \mid$
 $is\text{-measured}\text{-node } (Node3 \ a \ n1 \ n2 \ n3) \longleftrightarrow ((is\text{-measured}\text{-node } n1) \wedge$
 $(is\text{-measured}\text{-node } n2) \wedge (is\text{-measured}\text{-node } n3)) \wedge$
 $(a = (gmn \ n1) + (gmn \ n2) + (gmn \ n3))$

primrec $is\text{-measured}\text{-digit} :: ('e, 'a :: monoid\text{-add}) \text{Digit} \Rightarrow bool$ **where**
 $is\text{-measured}\text{-digit } (One \ a) = is\text{-measured}\text{-node } a \mid$
 $is\text{-measured}\text{-digit } (Two \ a \ b) =$
 $((is\text{-measured}\text{-node } a) \wedge (is\text{-measured}\text{-node } b)) \mid$
 $is\text{-measured}\text{-digit } (Three \ a \ b \ c) =$
 $((is\text{-measured}\text{-node } a) \wedge (is\text{-measured}\text{-node } b) \wedge (is\text{-measured}\text{-node } c)) \mid$
 $is\text{-measured}\text{-digit } (Four \ a \ b \ c \ d) = ((is\text{-measured}\text{-node } a) \wedge$
 $(is\text{-measured}\text{-node } b) \wedge (is\text{-measured}\text{-node } c) \wedge (is\text{-measured}\text{-node } d))$

primrec $is\text{-measured}\text{-ftree} :: ('e, 'a :: monoid\text{-add}) \text{FingerTreeStruc} \Rightarrow bool$ **where**
 $is\text{-measured}\text{-ftree } Empty \longleftrightarrow True \mid$
 $is\text{-measured}\text{-ftree } (Single \ n1) \longleftrightarrow (is\text{-measured}\text{-node } n1) \mid$
 $is\text{-measured}\text{-ftree } (Deep \ a \ l \ m \ r) \longleftrightarrow ((is\text{-measured}\text{-digit } l) \wedge$
 $(is\text{-measured}\text{-ftree } m) \wedge (is\text{-measured}\text{-digit } r)) \wedge$
 $(a = ((gmd \ l) + (gmft \ m) + (gmd \ r)))$

Structural invariant for finger trees

definition $ft\text{-invar } t == is\text{-leveln}\text{-ftree } 0 \ t \wedge is\text{-measured}\text{-ftree } t$

1.1.2 Abstraction to Lists

primrec $nodeTo\text{List} :: ('e, 'a) \text{Node} \Rightarrow ('e \times 'a) \text{list}$ **where**
 $nodeTo\text{List } (Tip \ e \ a) = [(e, a)]$
 $nodeTo\text{List } (Node2 \ - \ a \ b) = (nodeTo\text{List } a) @ (nodeTo\text{List } b) \mid$
 $nodeTo\text{List } (Node3 \ - \ a \ b \ c)$
 $= (nodeTo\text{List } a) @ (nodeTo\text{List } b) @ (nodeTo\text{List } c)$

primrec $digitTo\text{List} :: ('e, 'a) \text{Digit} \Rightarrow ('e \times 'a) \text{list}$ **where**
 $digitTo\text{List } (One \ a) = nodeTo\text{List } a \mid$
 $digitTo\text{List } (Two \ a \ b) = (nodeTo\text{List } a) @ (nodeTo\text{List } b) \mid$
 $digitTo\text{List } (Three \ a \ b \ c)$

$$\begin{aligned}
&= (\text{nodeToList } a) @ (\text{nodeToList } b) @ (\text{nodeToList } c) | \\
&\text{digitToList } (\text{Four } a \ b \ c \ d) \\
&= (\text{nodeToList } a) @ (\text{nodeToList } b) @ (\text{nodeToList } c) @ (\text{nodeToList } d)
\end{aligned}$$

List representation of a finger tree

primrec *toList* :: ('e, 'a) FingerTreeStruc \Rightarrow ('e \times 'a) list **where**
toList Empty = []
toList (Single a) = nodeToList a |
toList (Deep - pr m sf) = (digitToList pr) @ (toList m) @ (digitToList sf)

lemma *nodeToList-empty*: nodeToList nd \neq Nil
by (induct nd) auto

lemma *digitToList-empty*: digitToList d \neq Nil
by (cases d, auto simp add: nodeToList-empty)

Auxiliary lemmas

lemma *gmn-correct*:
assumes *is-measured-node nd*
shows *gmn nd = sum-list (map snd (nodeToList nd))*
by (insert assms, induct nd) (auto simp add: add.assoc)

lemma *gmd-correct*:
assumes *is-measured-digit d*
shows *gmd d = sum-list (map snd (digitToList d))*
by (insert assms, cases d, auto simp add: gmn-correct add.assoc)

lemma *gmft-correct: is-measured-ftree t*
 \implies (gmft t) = sum-list (map snd (toList t))
by (induct t, auto simp add: ft-invar-def gmd-correct gmn-correct add.assoc)
lemma *gmft-correct2: ft-invar t \implies (gmft t) = sum-list (map snd (toList t))*
by (simp only: ft-invar-def gmft-correct)

1.2 Operations

1.2.1 Empty tree

lemma *Empty-correct[simp]*:
toList Empty = []
ft-invar Empty
by (simp-all add: ft-invar-def)

Exactly the empty finger tree represents the empty list

lemma *toList-empty*: toList t = [] \longleftrightarrow t = Empty
by (induct t, auto simp add: nodeToList-empty digitToList-empty)

1.2.2 Annotation

Sum of annotations of all elements of a finger tree

definition *annot* :: ('e,'a::monoid-add) FingerTreeStruc \Rightarrow 'a
where *annot* t = gmft t

lemma *annot-correct*:

ft-invar t \implies *annot* t = sum-list (map snd (toList t))

using *gmft-correct*

unfolding *annot-def*

by (*simp* add: *gmft-correct2*)

1.2.3 Appending

Auxiliary functions to fill in the annotations

definition *deep*:: ('e,'a::monoid-add) Digit \Rightarrow ('e,'a) FingerTreeStruc
 \Rightarrow ('e,'a) Digit \Rightarrow ('e, 'a) FingerTreeStruc **where**

deep pr m sf = Deep ((gmd pr) + (gmft m) + (gmd sf)) pr m sf

definition *node2* **where**

node2 nd1 nd2 = Node2 ((gmn nd1)+(gmn nd2)) nd1 nd2

definition *node3* **where**

node3 nd1 nd2 nd3 = Node3 ((gmn nd1)+(gmn nd2)+(gmn nd3)) nd1 nd2 nd3

Append a node at the left end

fun *nlcons* :: ('e,'a::monoid-add) Node \Rightarrow ('e,'a) FingerTreeStruc
 \Rightarrow ('e,'a) FingerTreeStruc

where

— Recursively we append a node, if the digit is full we push down a node3

nlcons a Empty = Single a |

nlcons a (Single b) = deep (One a) Empty (One b) |

nlcons a (Deep - (One b) m sf) = deep (Two a b) m sf |

nlcons a (Deep - (Two b c) m sf) = deep (Three a b c) m sf |

nlcons a (Deep - (Three b c d) m sf) = deep (Four a b c d) m sf |

nlcons a (Deep - (Four b c d e) m sf)

= deep (Two a b) (nlcons (node3 c d e) m) sf

Append a node at the right end

fun *nrcons* :: ('e,'a::monoid-add) FingerTreeStruc

\Rightarrow ('e,'a) Node \Rightarrow ('e,'a) FingerTreeStruc **where**

— Recursively we append a node, if the digit is full we push down a node3

nrcons Empty a = Single a |

nrcons (Single b) a = deep (One b) Empty (One a) |

nrcons (Deep - pr m (One b)) a = deep pr m (Two b a) |

nrcons (Deep - pr m (Two b c)) a = deep pr m (Three b c a) |

nrcons (Deep - pr m (Three b c d)) a = deep pr m (Four b c d a) |

nrcons (Deep - pr m (Four b c d e)) a

= deep pr (nrcons m (node3 b c d)) (Two e a)

lemma *nlcons-inlevel*: [[*is-leveln-ftree* n t; *is-leveln-node* n nd]

\implies *is-leveln-ftree* n (nlcons nd t)

by (*induct* t arbitrary: n nd rule: *nlcons.induct*)

(*auto simp add: deep-def node3-def*)

lemma *nlcons-invmeas*: $\llbracket \text{is-measured-ftree } t; \text{is-measured-node } nd \rrbracket$
 $\implies \text{is-measured-ftree } (nlcons \ nd \ t)$
by (*induct t arbitrary: nd rule: nlcons.induct*)
(*auto simp add: deep-def node3-def*)

lemmas *nlcons-inv = nlcons-invlevel nlcons-invmeas*

lemma *nlcons-list*: $\text{toList } (nlcons \ a \ t) = (\text{nodeToList } a) \ @ \ (\text{toList } t)$
apply (*induct t arbitrary: a rule: nlcons.induct*)
apply (*auto simp add: deep-def toList-def node3-def*)
done

lemma *nrcons-invlevel*: $\llbracket \text{is-leveln-ftree } n \ t; \text{is-leveln-node } n \ nd \rrbracket$
 $\implies \text{is-leveln-ftree } n \ (nrcons \ t \ nd)$
apply (*induct t nd arbitrary: nd n rule: nrcons.induct*)
apply (*auto simp add: deep-def node3-def*)
done

lemma *nrcons-invmeas*: $\llbracket \text{is-measured-ftree } t; \text{is-measured-node } nd \rrbracket$
 $\implies \text{is-measured-ftree } (nrcons \ t \ nd)$
apply (*induct t nd arbitrary: nd rule: nrcons.induct*)
apply (*auto simp add: deep-def node3-def*)
done

lemmas *nrcons-inv = nrcons-invlevel nrcons-invmeas*

lemma *nrcons-list*: $\text{toList } (nrcons \ t \ a) = (\text{toList } t) \ @ \ (\text{nodeToList } a)$
apply (*induct t a arbitrary: a rule: nrcons.induct*)
apply (*auto simp add: deep-def toList-def node3-def*)
done

Append an element at the left end

definition *lcons* :: $(e \times 'a::\text{monoid-add})$
 $\Rightarrow (e, 'a) \text{FingerTreeStruc} \Rightarrow (e, 'a) \text{FingerTreeStruc}$ (**infixr** \triangleleft 65) **where**
 $a \triangleleft t = nlcons \ (\text{Tip } (fst \ a)) \ (snd \ a)) \ t$

lemma *lcons-correct*:
assumes *ft-invar t*
shows *ft-invar* $(a \triangleleft t)$ **and** $\text{toList } (a \triangleleft t) = a \ # \ (\text{toList } t)$
using *assms*
unfolding *ft-invar-def*
by (*simp-all add: lcons-def nlcons-list nlcons-invlevel nlcons-invmeas*)

lemma *lcons-inv:ft-invar t* $\implies \text{ft-invar } (a \triangleleft t)$
by (*rule lcons-correct*)

lemma *lcons-list[simp]*: $\text{toList } (a \triangleleft t) = a \ # \ (\text{toList } t)$

by (*simp add: lcons-def nlcons-list*)

Append an element at the right end

definition *rcons*

:: (*'e, 'a::monoid-add*) *FingerTreeStruc* \Rightarrow (*'e \times 'a*) \Rightarrow (*'e, 'a*) *FingerTreeStruc*
 (**infixl** \triangleright 65) **where**
t \triangleright *a* = *nrcons t (Tip (fst a) (snd a))*

lemma *rcons-correct*:

assumes *ft-invar t*
shows *ft-invar (t \triangleright a)* **and** *toList (t \triangleright a) = (toList t) @ [a]*
using *assms*
by (*auto simp add: nrcons-inv ft-invar-def rcons-def nrcons-list*)

lemma *rcons-inv:ft-invar t \implies ft-invar (t \triangleright a)*

by (*rule rcons-correct*)

lemma *rcons-list[simp]: toList (t \triangleright a) = (toList t) @ [a]*

by(*auto simp add: nrcons-list rcons-def*)

1.2.4 Convert list to tree

primrec *toTree* :: (*'e \times 'a::monoid-add*) *list* \Rightarrow (*'e, 'a*) *FingerTreeStruc* **where**

toTree [] = Empty
toTree (a#xs) = a \triangleleft (toTree xs)

lemma *toTree-correct[simp]*:

ft-invar (toTree l)
toList (toTree l) = l
apply (*induct l*)
apply (*simp add: ft-invar-def*)
apply *simp*
apply (*simp add: toTree-def lcons-list lcons-inv*)
apply (*simp add: toTree-def lcons-list lcons-inv*)
done

Note that this lemma is a completeness statement of our implementation, as it can be read as: „All lists of elements have a valid representation as a finger tree.”

1.2.5 Detaching leftmost/rightmost element

primrec *digitToTree* :: (*'e, 'a::monoid-add*) *Digit* \Rightarrow (*'e, 'a*) *FingerTreeStruc*
where

digitToTree (One a) = Single a
digitToTree (Two a b) = deep (One a) Empty (One b)
digitToTree (Three a b c) = deep (Two a b) Empty (One c)
digitToTree (Four a b c d) = deep (Two a b) Empty (Two c d)

primrec *nodeToDigit* :: ('e,'a) Node ⇒ ('e,'a) Digit **where**
nodeToDigit (Tip e a) = One (Tip e a)|
nodeToDigit (Node2 - a b) = Two a b|
nodeToDigit (Node3 - a b c) = Three a b c

fun *nlistToDigit* :: ('e,'a) Node list ⇒ ('e,'a) Digit **where**
nlistToDigit [a] = One a |
nlistToDigit [a,b] = Two a b |
nlistToDigit [a,b,c] = Three a b c |
nlistToDigit [a,b,c,d] = Four a b c d

primrec *digitToNlist* :: ('e,'a) Digit ⇒ ('e,'a) Node list **where**
digitToNlist (One a) = [a] |
digitToNlist (Two a b) = [a,b] |
digitToNlist (Three a b c) = [a,b,c] |
digitToNlist (Four a b c d) = [a,b,c,d]

Auxiliary function to unwrap a Node element

primrec *n-unwrap*:: ('e,'a) Node ⇒ ('e × 'a) **where**
n-unwrap (Tip e a) = (e,a)|
n-unwrap (Node2 - a b) = undefined|
n-unwrap (Node3 - a b c) = undefined

type-synonym ('e,'a) ViewnRes = (('e,'a) Node × ('e,'a) FingerTreeStruc) option

lemma *viewnres-cases*:

fixes *r* :: ('e,'a) ViewnRes
obtains (Nil) *r=Some* |
(Cons) *a t where r=Some* (a,t)
by (cases *r*) *auto*

lemma *viewnres-split*:

P (case-option *f1* (case-prod *f2*) *x*) =
((*x* = None → *P f1*) ∧ (∀ *a b. x* = Some (*a,b*) → *P (f2 a b)*))
by (auto split: option.split prod.split)

Detach the leftmost node. Return *None* on empty finger tree.

fun *viewLn* :: ('e,'a::monoid-add) FingerTreeStruc ⇒ ('e,'a) ViewnRes **where**
viewLn Empty = None|
viewLn (Single a) = Some (a, Empty)|
viewLn (Deep - (Two a b) m sf) = Some (a, (deep (One b) m sf))|
viewLn (Deep - (Three a b c) m sf) = Some (a, (deep (Two b c) m sf))|
viewLn (Deep - (Four a b c d) m sf) = Some (a, (deep (Three b c d) m sf))|
viewLn (Deep - (One a) m sf) =
(case *viewLn m* of
None ⇒ Some (a, (digitToTree sf)) |
Some (b, m2) ⇒ Some (a, (deep (nodeToDigit b) m2 sf)))

Detach the rightmost node. Return *None* on empty finger tree.

```

fun viewRn :: ('e,'a::monoid-add) FingerTreeStruc ⇒ ('e,'a) ViewnRes where
  viewRn Empty = None |
  viewRn (Single a) = Some (a, Empty) |
  viewRn (Deep - pr m (Two a b)) = Some (b, (deep pr m (One a))) |
  viewRn (Deep - pr m (Three a b c)) = Some (c, (deep pr m (Two a b))) |
  viewRn (Deep - pr m (Four a b c d)) = Some (d, (deep pr m (Three a b c))) |
  viewRn (Deep - pr m (One a)) =
    (case viewRn m of
      None ⇒ Some (a, (digitToTree pr))|
      Some (b, m2) ⇒ Some (a, (deep pr m2 (nodeToDigit b))))

```

lemma

```

  digitToTree-inv: is-leveln-digit n d ⇒ is-leveln-ftree n (digitToTree d)
  is-measured-digit d ⇒ is-measured-ftree (digitToTree d)
apply (cases d,auto simp add: deep-def)
apply (cases d,auto simp add: deep-def)
done

```

lemma digitToTree-list: toList (digitToTree d) = digitToList d
by (cases d) (auto simp add: deep-def)

lemma nodeToDigit-inv:

```

  is-leveln-node (Suc n) nd ⇒ is-leveln-digit n (nodeToDigit nd)
  is-measured-node nd ⇒ is-measured-digit (nodeToDigit nd)
by (cases nd, auto) (cases nd, auto)

```

lemma nodeToDigit-list: digitToList (nodeToDigit nd) = nodeToList nd
by (cases nd,auto)

lemma viewLn-empty: $t \neq \text{Empty} \iff (\text{viewLn } t) \neq \text{None}$

proof (cases t)

case Empty **thus** ?thesis **by** simp

next

case (Single Node) **thus** ?thesis **by** simp

next

case (Deep a l x r) **thus** ?thesis

apply(auto)

apply(case-tac l)

apply(auto)

apply(cases viewLn x)

apply(auto)

done

qed

lemma viewLn-inv: [

```

is-measured-ftree t; is-leveln-ftree n t; viewLn t = Some (nd, s)
]  $\implies$  is-measured-ftree s  $\wedge$  is-measured-node nd  $\wedge$ 
  is-leveln-ftree n s  $\wedge$  is-leveln-node n nd
apply(induct t arbitrary: n nd s rule: viewLn.induct)
apply(simp add: viewLn-empty)
apply(simp)
apply(auto simp add: deep-def)[1]
apply(auto simp add: deep-def)[1]
apply(auto simp add: deep-def)[1]
proof -
fix ux a m sf n nd s
assume av:  $\bigwedge n nd s.$ 
   $\llbracket$ is-measured-ftree m; is-leveln-ftree n m; viewLn m = Some (nd, s) $\rrbracket$ 
   $\implies$  is-measured-ftree s  $\wedge$ 
    is-measured-node nd  $\wedge$  is-leveln-ftree n s  $\wedge$  is-leveln-node n nd
    is-measured-ftree (Deep ux (One a) m sf)
    is-leveln-ftree n (Deep ux (One a) m sf)
    viewLn (Deep ux (One a) m sf) = Some (nd, s)
thus is-measured-ftree s  $\wedge$ 
  is-measured-node nd  $\wedge$  is-leveln-ftree n s  $\wedge$  is-leveln-node n nd
proof (cases viewLn m rule: viewnres-cases)
  case Nil
  with av(4) have v1: nd = a s = digitToTree sf
  by auto
  from v1 av(2,3) show is-measured-ftree s  $\wedge$ 
    is-measured-node nd  $\wedge$  is-leveln-ftree n s  $\wedge$  is-leveln-node n nd
  apply(auto)
  apply(auto simp add: digitToTree-inv)
  done
next
  case (Cons b m2)
  with av(4) have v2: nd = a s = (deep (nodeToDigit b) m2 sf)
  apply (auto simp add: deep-def)
  done
  note myiv = av(1)[of Suc n b m2]
  from v2 av(2,3) have is-measured-ftree m  $\wedge$  is-leveln-ftree (Suc n) m
  apply(simp)
  done
  hence bv: is-measured-ftree m2  $\wedge$ 
    is-measured-node b  $\wedge$  is-leveln-ftree (Suc n) m2  $\wedge$  is-leveln-node (Suc n) b
  using myiv Cons
  apply(simp)
  done
  with av(2,3) v2 show is-measured-ftree s  $\wedge$ 
    is-measured-node nd  $\wedge$  is-leveln-ftree n s  $\wedge$  is-leveln-node n nd
  apply(auto simp add: deep-def nodeToDigit-inv)
  done
qed
qed

```

```

lemma viewLn-list: viewLn t = Some (nd, s)
   $\implies$  toList t = (nodeToList nd) @ (toList s)
  supply [[simp del: defined-all]]
  apply(induct t arbitrary: nd s rule: viewLn.induct)
  apply(simp)
  apply(simp)
  apply(simp)
  apply(simp add: deep-def)
  apply(auto simp add: toList-def)[1]
  apply(simp)
  apply(simp add: deep-def)
  apply(auto simp add: toList-def)[1]
  apply(simp)
  apply(simp add: deep-def)
  apply(auto simp add: toList-def)[1]
  apply(simp)
  subgoal premises prems for a m sf nd s
    using prems
  proof (cases viewLn m rule: viewnres-cases)
    case Nil
      hence av: m = Empty by (metis viewLn-empty)
      from av prems
      show nodeToList a @ toList m @ digitToList sf = nodeToList nd @ toList s
        by (auto simp add: digitToTree-list)
    next
      case (Cons b m2)
      with prems have bv: nd = a s = (deep (nodeToDigit b) m2 sf)
        by (auto simp add: deep-def)
      with Cons prems
      show nodeToList a @ toList m @ digitToList sf = nodeToList nd @ toList s
        apply(simp)
        apply(simp add: deep-def)
        apply(simp add: deep-def nodeToDigit-list)
        done
    qed
  done

```

```

lemma viewRn-empty: t  $\neq$  Empty  $\iff$  (viewRn t)  $\neq$  None
proof (cases t)
  case Empty thus ?thesis by simp
next
  case (Single Node) thus ?thesis by simp
next
  case (Deep a l x r) thus ?thesis
    apply(auto)
    apply(case-tac r)
    apply(auto)
    apply(cases viewRn x)

```

apply(*auto*)
done
qed

lemma *viewRn-inv*: \llbracket
is-measured-ftree *t*; *is-leveln-ftree* *n* *t*; *viewRn* *t* = *Some* (*nd*, *s*)
 $\rrbracket \implies$ *is-measured-ftree* *s* \wedge *is-measured-node* *nd* \wedge
is-leveln-ftree *n* *s* \wedge *is-leveln-node* *n* *nd*
apply(*induct* *t* *arbitrary*: *n* *nd* *s* *rule*: *viewRn.induct*)
apply(*simp* *add*: *viewRn-empty*)
apply(*simp*)
apply(*auto* *simp* *add*: *deep-def*)[1]
apply(*auto* *simp* *add*: *deep-def*)[1]
apply(*auto* *simp* *add*: *deep-def*)[1]

proof –

fix *ux* *a* *m* *pr* *n* *nd* *s*
assume *av*: $\bigwedge n$ *nd* *s*.
 \llbracket *is-measured-ftree* *m*; *is-leveln-ftree* *n* *m*; *viewRn* *m* = *Some* (*nd*, *s*) \rrbracket
 \implies *is-measured-ftree* *s* \wedge
is-measured-node *nd* \wedge *is-leveln-ftree* *n* *s* \wedge *is-leveln-node* *n* *nd*
is-measured-ftree (*Deep* *ux* *pr* *m* (*One* *a*))
is-leveln-ftree *n* (*Deep* *ux* *pr* *m* (*One* *a*))
viewRn (*Deep* *ux* *pr* *m* (*One* *a*)) = *Some* (*nd*, *s*)

thus *is-measured-ftree* *s* \wedge
is-measured-node *nd* \wedge *is-leveln-ftree* *n* *s* \wedge *is-leveln-node* *n* *nd*

proof (*cases* *viewRn* *m* *rule*: *viewnres-cases*)

case *Nil*

with *av*(4) **have** *v1*: *nd* = *a* *s* = *digitToTree* *pr*

by *auto*

from *v1* *av*(2,3) **show** *is-measured-ftree* *s* \wedge

is-measured-node *nd* \wedge *is-leveln-ftree* *n* *s* \wedge *is-leveln-node* *n* *nd*

apply(*auto*)

apply(*auto* *simp* *add*: *digitToTree-inv*)

done

next

case (*Cons* *b* *m2*)

with *av*(4) **have** *v2*: *nd* = *a* *s* = (*deep* *pr* *m2* (*nodeToDigit* *b*))

apply (*auto* *simp* *add*: *deep-def*)

done

note *myiv* = *av*(1)[*of* *Suc* *n* *b* *m2*]

from *v2* *av*(2,3) **have** *is-measured-ftree* *m* \wedge *is-leveln-ftree* (*Suc* *n*) *m*

apply(*simp*)

done

hence *bv*: *is-measured-ftree* *m2* \wedge

is-measured-node *b* \wedge *is-leveln-ftree* (*Suc* *n*) *m2* \wedge *is-leveln-node* (*Suc* *n*) *b*

using *myiv* *Cons*

apply(*simp*)

done

with *av*(2,3) *v2* **show** *is-measured-ftree* *s* \wedge

```

    is-measured-node nd ∧ is-leveln-ftree n s ∧ is-leveln-node n nd
  apply(auto simp add: deep-def nodeToDigit-inv)
done
qed
qed

lemma viewRn-list: viewRn t = Some (nd, s)
  ⇒ toList t = (toList s) @ (nodeToList nd)
  supply [[simp proc del: defined-all]]
  apply(induct t arbitrary: nd s rule: viewRn.induct)
  apply(simp)
  apply(simp)
  apply(simp)
  apply(simp add: deep-def)
  apply(auto simp add: toList-def)[1]
  apply(simp)
  apply(simp add: deep-def)
  apply(auto simp add: toList-def)[1]
  apply(simp)
  apply(simp add: deep-def)
  apply(auto simp add: toList-def)[1]
  apply(simp)
  subgoal premises prems for pr m a nd s
  proof (cases viewRn m rule: viewnres-cases)
  case Nil
  from Nil have av: m = Empty by (metis viewRn-empty)
  from av prems
  show digitToList pr @ toList m @ nodeToList a = toList s @ nodeToList nd
  by (auto simp add: digitToTree-list)
next
  case (Cons b m2)
  with prems have bv: nd = a s = (deep pr m2 (nodeToDigit b))
  apply(auto simp add: deep-def) done
  with Cons prems
  show digitToList pr @ toList m @ nodeToList a = toList s @ nodeToList nd
  apply(simp)
  apply(simp add: deep-def)
  apply(simp add: deep-def nodeToDigit-list)
  done
qed
done

```

type-synonym ('e,'a) viewres = (('e × 'a) × ('e,'a) FingerTreeStruc) option

Detach the leftmost element. Return *None* on empty finger tree.

definition viewL :: ('e,'a::monoid-add) FingerTreeStruc ⇒ ('e,'a) viewres
 where
 viewL t = (case viewLn t of
 None ⇒ None |

$(\text{Some } (a, t2)) \Rightarrow \text{Some } ((n\text{-unwrap } a), t2))$

lemma *viewL-correct*:

assumes *INV*: *ft-invar t*

shows

$(t = \text{Empty} \Rightarrow \text{viewL } t = \text{None})$

$(t \neq \text{Empty} \Rightarrow (\exists a s. \text{viewL } t = \text{Some } (a, s) \wedge \text{ft-invar } s$
 $\wedge \text{toList } t = a \# \text{toList } s))$

proof –

assume $t = \text{Empty}$ **thus** $\text{viewL } t = \text{None}$ **by** (*simp add: viewL-def*)

next

assume *NE*: $t \neq \text{Empty}$

from *INV* **have** *INV'*: *is-leveln-ftree 0 t is-measured-ftree t*

by (*simp-all add: ft-invar-def*)

from *NE* **have** *v1*: $\text{viewLn } t \neq \text{None}$ **by** (*auto simp add: viewLn-empty*)

then obtain *nd s* **where** *vn*: $\text{viewLn } t = \text{Some } (nd, s)$

by (*cases viewLn t*) (*auto*)

from this obtain *a* **where** *v1*: $\text{viewL } t = \text{Some } (a, s)$

by (*auto simp add: viewL-def*)

from *INV'* *vn* **have**

v2: $\text{is-measured-ftree } s \wedge \text{is-leveln-ftree } 0 s$

$\wedge \text{is-leveln-node } 0 nd \wedge \text{is-measured-node } nd$

$\text{toList } t = (\text{nodeToList } nd) \text{ @ } (\text{toList } s)$

by (*auto simp add: viewLn-inv[of t 0 nd s] viewLn-list[of t]*)

with *v1 vn* **have** *v3*: $\text{nodeToList } nd = [a]$

apply (*auto simp add: viewL-def*)

apply (*induct nd*)

apply (*simp-all (no-asm-use)*)

done

with *v1 v2*

show $\exists a s. \text{viewL } t = \text{Some } (a, s) \wedge \text{ft-invar } s \wedge \text{toList } t = a \# \text{toList } s$

by (*auto simp add: ft-invar-def*)

qed

lemma *viewL-correct-empty[simp]*: $\text{viewL } \text{Empty} = \text{None}$

by (*simp add: viewL-def*)

lemma *viewL-correct-nonEmpty*:

assumes *ft-invar t t* $t \neq \text{Empty}$

obtains *a s* **where**

$\text{viewL } t = \text{Some } (a, s) \wedge \text{ft-invar } s \wedge \text{toList } t = a \# \text{toList } s$

using *assms viewL-correct* **by** *blast*

Detach the rightmost element. Return *None* on empty finger tree.

definition *viewR* :: $('e, 'a :: \text{monoid-add}) \text{ FingerTreeStruc} \Rightarrow ('e, 'a) \text{ views}$

where

$\text{viewR } t = (\text{case viewRn } t \text{ of}$

$\text{None} \Rightarrow \text{None} \mid$

$(\text{Some } (a, t2)) \Rightarrow \text{Some } ((n\text{-unwrap } a), t2))$

```

lemma viewR-correct:
  assumes INV: ft-invar t
  shows
    (t = Empty  $\implies$  viewR t = None)
    (t  $\neq$  Empty  $\implies$  ( $\exists$  a s. viewR t = Some (a, s)  $\wedge$  ft-invar s
       $\wedge$  toList t = toList s @ [a]))
proof –
  assume t=Empty thus viewR t = None by (simp add: viewR-def)
next
  assume NE: t  $\neq$  Empty
  from INV have INV': is-leveln-ftree 0 t is-measured-ftree t
    unfolding ft-invar-def by simp-all
  from NE have v1: viewRn t  $\neq$  None by (auto simp add: viewRn-empty)
  then obtain nd s where vn: viewRn t = Some (nd, s)
    by (cases viewRn t) (auto)
  from this obtain a where v1: viewR t = Some (a, s)
    by (auto simp add: viewR-def)
  from INV' vn have
    v2: is-measured-ftree s  $\wedge$  is-leveln-ftree 0 s
       $\wedge$  is-leveln-node 0 nd  $\wedge$  is-measured-node nd
      toList t = (toList s) @ (nodeToList nd)
    by (auto simp add: viewRn-inv[of t 0 nd s] viewRn-list[of t])
  with v1 vn have v3: nodeToList nd = [a]
    apply (auto simp add: viewR-def)
    apply (induct nd)
    apply (simp-all (no-asm-use))
  done
  with v1 v2
  show  $\exists$  a s. viewR t = Some (a, s)  $\wedge$  ft-invar s  $\wedge$  toList t = toList s @ [a]
    unfolding ft-invar-def by auto
qed

```

```

lemma viewR-correct-empty[simp]: viewR Empty = None
  unfolding viewR-def by simp

```

```

lemma viewR-correct-nonEmpty:
  assumes ft-invar t t  $\neq$  Empty
  obtains a s where
    viewR t = Some (a, s) ft-invar s  $\wedge$  toList t = toList s @ [a]
  using assms viewR-correct by blast

```

Finger trees viewed as a double-ended queue. The head and tail functions here are only defined for non-empty queues, while the view-functions were also defined for empty finger trees.

Check for emptiness

```

definition isEmpty :: ('e,'a) FingerTreeStruc  $\Rightarrow$  bool where
  [code del]: isEmpty t = (t = Empty)

```


lemma *isEmpty-correct*: $isEmpty\ t \longleftrightarrow toList\ t = []$
unfolding *isEmpty-def* **by** (*simp add: toList-empty*)
— Avoid comparison with (=), and thus unnecessary equality-class parameter on element types in generated code
lemma [*code*]: $isEmpty\ t = (case\ t\ of\ Empty \Rightarrow True \mid - \Rightarrow False)$
apply (*cases t*)
apply (*auto simp add: isEmpty-def*)
done

Leftmost element

definition *head* :: (*'e,'a::monoid-add*) *FingerTreeStruc* $\Rightarrow 'e \times 'a$ **where**
head t = (*case viewL t of (Some (a, -)) \Rightarrow a*)

lemma *head-correct*:

assumes *ft-invar t t \neq Empty*
shows *head t = hd (toList t)*

proof —

from *assms viewL-correct*
obtain *a s where*

v1:viewL t = Some (a, s) \wedge ft-invar s \wedge toList t = a $\#$ toList s **by** *blast*

hence *v2: head t = a* **by** (*auto simp add: head-def*)

from *v1* **have** *hd (toList t) = a* **by** *simp*

with *v2* **show** *?thesis* **by** *simp*

qed

All but the leftmost element

definition *tail*

:: (*'e,'a::monoid-add*) *FingerTreeStruc* $\Rightarrow ('e,'a)$ *FingerTreeStruc*
where

tail t = (*case viewL t of (Some (-, m)) \Rightarrow m*)

lemma *tail-correct*:

assumes *ft-invar t t \neq Empty*

shows *toList (tail t) = tl (toList t)* **and** *ft-invar (tail t)*

proof —

from *assms viewL-correct*
obtain *a s where*

v1:viewL t = Some (a, s) \wedge ft-invar s \wedge toList t = a $\#$ toList s **by** *blast*

hence *v2: tail t = s* **by** (*auto simp add: tail-def*)

from *v1* **have** *tl (toList t) = toList s* **by** *simp*

with *v1 v2* **show**

toList (tail t) = tl (toList t)

ft-invar (tail t)

by *simp-all*

qed

Rightmost element

definition *headR* :: (*'e,'a::monoid-add*) *FingerTreeStruc* $\Rightarrow 'e \times 'a$ **where**

headR t = (*case viewR t of (Some (a, -)) \Rightarrow a*)

lemma *headR-correct*:

assumes *ft-invar t t \neq Empty*

shows $\text{headR } t = \text{last } (\text{toList } t)$
proof –
from *assms viewR-correct*
obtain $a \ s$ **where**
 $v1:\text{viewR } t = \text{Some } (a, s) \wedge \text{ft-invar } s \wedge \text{toList } t = \text{toList } s \ @ \ [a]$ **by** *blast*
hence $v2:\text{headR } t = a$ **by** (*auto simp add: headR-def*)
with $v1$ **show** *?thesis* **by** *auto*
qed

All but the rightmost element

definition *tailR*
 $:: ('e, 'a::\text{monoid-add}) \text{FingerTreeStruc} \Rightarrow ('e, 'a) \text{FingerTreeStruc}$
where
 $\text{tailR } t = (\text{case viewR } t \text{ of } (\text{Some } (-, m)) \Rightarrow m)$
lemma *tailR-correct*:
assumes $\text{ft-invar } t \neq \text{Empty}$
shows $\text{toList } (\text{tailR } t) = \text{butlast } (\text{toList } t)$ **and** $\text{ft-invar } (\text{tailR } t)$
proof –
from *assms viewR-correct*
obtain $a \ s$ **where**
 $v1:\text{viewR } t = \text{Some } (a, s) \wedge \text{ft-invar } s \wedge \text{toList } t = \text{toList } s \ @ \ [a]$ **by** *blast*
hence $v2:\text{tailR } t = s$ **by** (*auto simp add: tailR-def*)
with $v1$ **show** $\text{toList } (\text{tailR } t) = \text{butlast } (\text{toList } t)$ **and** $\text{ft-invar } (\text{tailR } t)$
by *auto*
qed

1.2.6 Concatenation

primrec *lconsNlist* $:: ('e, 'a::\text{monoid-add}) \text{Node list} \Rightarrow ('e, 'a) \text{FingerTreeStruc} \Rightarrow ('e, 'a) \text{FingerTreeStruc}$ **where**
 $\text{lconsNlist } [] \ t = t \mid$
 $\text{lconsNlist } (x\#xs) \ t = \text{nlcons } x \ (\text{lconsNlist } xs \ t)$
primrec *rconsNlist* $:: ('e, 'a::\text{monoid-add}) \text{FingerTreeStruc} \Rightarrow ('e, 'a) \text{Node list} \Rightarrow ('e, 'a) \text{FingerTreeStruc}$ **where**
 $\text{rconsNlist } t \ [] = t \mid$
 $\text{rconsNlist } t \ (x\#xs) = \text{rconsNlist } (\text{nrcons } t \ x) \ xs$

fun *nodes* $:: ('e, 'a::\text{monoid-add}) \text{Node list} \Rightarrow ('e, 'a) \text{Node list}$ **where**
 $\text{nodes } [a, b] = [\text{node2 } a \ b] \mid$
 $\text{nodes } [a, b, c] = [\text{node3 } a \ b \ c] \mid$
 $\text{nodes } [a, b, c, d] = [\text{node2 } a \ b, \text{node2 } c \ d] \mid$
 $\text{nodes } (a\#b\#c\#xs) = (\text{node3 } a \ b \ c) \# (\text{nodes } xs)$

Recursively we concatenate two FingerTreeStrucs while we keep the inner Nodes in a list

fun *app3* $:: ('e, 'a::\text{monoid-add}) \text{FingerTreeStruc} \Rightarrow ('e, 'a) \text{Node list} \Rightarrow ('e, 'a) \text{FingerTreeStruc} \Rightarrow ('e, 'a) \text{FingerTreeStruc}$ **where**
 $\text{app3 } \text{Empty } xs \ t = \text{lconsNlist } xs \ t \mid$
 $\text{app3 } t \ xs \ \text{Empty} = \text{rconsNlist } t \ xs \mid$

$app3 (Single\ x)\ xs\ t = nlcons\ x\ (lconsNlist\ xs\ t) \mid$
 $app3\ t\ xs\ (Single\ x) = nrcons\ (rconsNlist\ t\ xs)\ x \mid$
 $app3\ (Deep\ -\ pr1\ m1\ sf1)\ ts\ (Deep\ -\ pr2\ m2\ sf2) =$
 $deep\ pr1\ (app3\ m1$
 $\quad (nodes\ ((digitToNlist\ sf1)\ @\ ts\ @\ (digitToNlist\ pr2)))\ m2)\ sf2$

lemma *lconsNlist-inv*:

assumes *is-leveln-ftree* $n\ t$

and *is-measured-ftree* t

and $\forall x \in set\ xs.\ (is-leveln-node\ n\ x \wedge is-measured-node\ x)$

shows

is-leveln-ftree $n\ (lconsNlist\ xs\ t) \wedge is-measured-ftree\ (lconsNlist\ xs\ t)$

by (*insert assms, induct xs, auto simp add: nlcons-invlevel nlcons-invmeas*)

lemma *rconsNlist-inv*:

assumes *is-leveln-ftree* $n\ t$

and *is-measured-ftree* t

and $\forall x \in set\ xs.\ (is-leveln-node\ n\ x \wedge is-measured-node\ x)$

shows

is-leveln-ftree $n\ (rconsNlist\ t\ xs) \wedge is-measured-ftree\ (rconsNlist\ t\ xs)$

by (*insert assms, induct xs arbitrary: t,*

auto simp add: nrcons-invlevel nrcons-invmeas)

lemma *nodes-inv*:

assumes $\forall x \in set\ ts.\ is-leveln-node\ n\ x \wedge is-measured-node\ x$

and $length\ ts \geq 2$

shows $\forall x \in set\ (nodes\ ts).\ is-leveln-node\ (Suc\ n)\ x \wedge is-measured-node\ x$

proof (*insert assms, induct ts rule: nodes.induct*)

case (1 $a\ b$)

thus *?case by (simp add: node2-def)*

next

case (2 $a\ b\ c$)

thus *?case by (simp add: node3-def)*

next

case (3 $a\ b\ c\ d$)

thus *?case by (simp add: node2-def)*

next

case (4 $a\ b\ c\ v\ vb\ vc$)

thus *?case by (simp add: node3-def)*

next

show $\llbracket \forall x \in set\ [].\ is-leveln-node\ n\ x \wedge is-measured-node\ x; 2 \leq length\ [] \rrbracket$

$\implies \forall x \in set\ (nodes\ []).\ is-leveln-node\ (Suc\ n)\ x \wedge is-measured-node\ x$

by *simp*

next

show

$\bigwedge v.\ \llbracket \forall x \in set\ [v].\ is-leveln-node\ n\ x \wedge is-measured-node\ x; 2 \leq length\ [v] \rrbracket$

$\implies \forall x \in set\ (nodes\ [v]).\ is-leveln-node\ (Suc\ n)\ x \wedge is-measured-node\ x$

by *simp*

qed

lemma *nodes-inv2*:

assumes *is-leveln-digit n sf1*
and *is-measured-digit sf1*
and *is-leveln-digit n pr2*
and *is-measured-digit pr2*
and $\forall x \in \text{set } ts. \text{is-leveln-node } n \ x \wedge \text{is-measured-node } x$
shows
 $\forall x \in \text{set } (\text{nodes } (\text{digitToNlist } sf1 \ @ \ ts \ @ \ \text{digitToNlist } pr2)).$
 $\text{is-leveln-node } (\text{Suc } n) \ x \wedge \text{is-measured-node } x$

proof –

have *v1*: $\forall x \in \text{set } (\text{digitToNlist } sf1 \ @ \ ts \ @ \ \text{digitToNlist } pr2).$
 $\text{is-leveln-node } n \ x \wedge \text{is-measured-node } x$

using *assms*
apply (*simp add: digitToNlist-def*)
apply (*cases sf1*)
apply (*cases pr2*)
apply *simp-all*
apply (*cases pr2*)
apply (*simp-all*)
apply (*cases pr2*)
apply (*simp-all*)
apply (*cases pr2*)
apply (*simp-all*)
done

have *v2*: $\text{length } (\text{digitToNlist } sf1 \ @ \ ts \ @ \ \text{digitToNlist } pr2) \geq 2$
apply (*cases sf1*)
apply (*cases pr2*)
apply *simp-all*
done

thus *?thesis*
using *v1 nodes-inv[of digitToNlist sf1 @ ts @ digitToNlist pr2]*
by *blast*

qed

lemma *app3-inv*:

assumes *is-leveln-ftree n t1*
and *is-leveln-ftree n t2*
and *is-measured-ftree t1*
and *is-measured-ftree t2*
and $\forall x \in \text{set } xs. (\text{is-leveln-node } n \ x \wedge \text{is-measured-node } x)$
shows $\text{is-leveln-ftree } n \ (\text{app3 } t1 \ xs \ t2) \wedge \text{is-measured-ftree } (\text{app3 } t1 \ xs \ t2)$

proof (*insert assms, induct t1 xs t2 arbitrary; n rule: app3.induct*)
case (*1 xs t n*)
thus *?case using lconsNlist-inv by simp*
next
case *2-1*
thus *?case by (simp add: rconsNlist-inv)*
next

```

case 2-2
thus ?case by (simp add: lconsNlist-inv rconsNlist-inv)
next
case 3-1
thus ?case by (simp add: lconsNlist-inv nlcons-invlevel nlcons-invmeas )
next
case 3-2
thus ?case
  by (simp only: app3.simps)
  (simp add: lconsNlist-inv nlcons-invlevel nlcons-invmeas)
next
case 4
thus ?case
  by (simp only: app3.simps)
  (simp add: rconsNlist-inv nrcons-invlevel nrcons-invmeas)
next
case (5 uu pr1 m1 sf1 ts uv pr2 m2 sf2 n)
thus ?case
proof -
  have v1: is-leveln-ftree (Suc n) m1
    and v2: is-leveln-ftree (Suc n) m2
    using 5.prem1 by (simp-all add: is-leveln-ftree-def)
  have v3: is-measured-ftree m1
    and v4: is-measured-ftree m2
    using 5.prem2 by (simp-all add: is-measured-ftree-def)
  have v5: is-leveln-digit n sf1
    is-measured-digit sf1
    is-leveln-digit n pr2
    is-measured-digit pr2
     $\forall x \in \text{set } ts. \text{is-leveln-node } n \ x \wedge \text{is-measured-node } x$ 
    using 5.prem3
    by (simp-all add: is-leveln-ftree-def is-measured-ftree-def)
  note v6 = nodes-inv2[OF v5]
  note v7 = 5.hyps[OF v1 v2 v3 v4 v6]
  have v8: is-leveln-digit n sf2
    is-measured-digit sf2
    is-leveln-digit n pr1
    is-measured-digit pr1
    using 5.prem4
    by (simp-all add: is-leveln-ftree-def is-measured-ftree-def)

  show ?thesis using v7 v8
    by (simp add: is-leveln-ftree-def is-measured-ftree-def deep-def)
qed
qed

```

```

primrec nlistToList:: (('e, 'a) Node) list  $\Rightarrow$  ('e  $\times$  'a) list where
  nlistToList [] = []
  nlistToList (x#xs) = (nodeToList x) @ (nlistToList xs)

```

lemma nodes-list: $\text{length } xs \geq 2 \implies \text{nlistToList } (\text{nodes } xs) = \text{nlistToList } xs$
by (*induct xs rule: nodes.induct*) (*auto simp add: node2-def node3-def*)

lemma nlistToList-app:
 $\text{nlistToList } (xs@ys) = (\text{nlistToList } xs) @ (\text{nlistToList } ys)$
by (*induct xs arbitrary: ys, simp-all*)

lemma nlistListLCons: $\text{toList } (\text{lconsNlist } xs t) = (\text{nlistToList } xs) @ (\text{toList } t)$
by (*induct xs*) (*auto simp add: nlcons-list*)

lemma nlistListRCons: $\text{toList } (\text{rconsNlist } t xs) = (\text{toList } t) @ (\text{nlistToList } xs)$
by (*induct xs arbitrary: t*) (*auto simp add: nrcons-list*)

lemma app3-list-lem1:
 $\text{nlistToList } (\text{nodes } (\text{digitToNlist } sf1 @ ts @ \text{digitToNlist } pr2)) =$
 $\text{digitToNlist } sf1 @ \text{nlistToList } ts @ \text{digitToNlist } pr2$

proof –

have $\text{len1: length } (\text{digitToNlist } sf1 @ ts @ \text{digitToNlist } pr2) \geq 2$
by (*cases sf1, cases pr2, simp-all*)

have $(\text{nlistToList } (\text{digitToNlist } sf1 @ ts @ \text{digitToNlist } pr2)) =$
 $(\text{digitToNlist } sf1 @ \text{nlistToList } ts @ \text{digitToNlist } pr2)$

apply (*cases sf1, cases pr2*)

apply (*simp-all add: nlistToList-app*)

apply (*cases pr2, auto*)

apply (*cases pr2, auto*)

apply (*cases pr2, auto*)

done

with nodes-list[OF len1] show ?thesis by simp

qed

lemma app3-list:
 $\text{toList } (\text{app3 } t1 xs t2) = (\text{toList } t1) @ (\text{nlistToList } xs) @ (\text{toList } t2)$
apply (*induct t1 xs t2 rule: app3.induct*)
apply (*simp-all add: nlistListLCons nlistListRCons nlcons-list nrcons-list*)
apply (*simp add: app3-list-lem1 deep-def*)
done

definition app
 $:: ('e, 'a::\text{monoid-add}) \text{FingerTreeStruc} \implies ('e, 'a) \text{FingerTreeStruc}$
 $\implies ('e, 'a) \text{FingerTreeStruc}$
where $\text{app } t1 t2 = \text{app3 } t1 [] t2$

lemma app-correct:
assumes *ft-invar t1 ft-invar t2*
shows $\text{toList } (\text{app } t1 t2) = (\text{toList } t1) @ (\text{toList } t2)$

```

    and ft-invar (app t1 t2)
  using assms
  by (auto simp add: app3-inv app3-list ft-invar-def app-def)

lemma app-inv:  $\llbracket \text{ft-invar } t1; \text{ft-invar } t2 \rrbracket \implies \text{ft-invar } (\text{app } t1 \ t2)$ 
  by (auto simp add: app3-inv ft-invar-def app-def)

lemma app-list[simp]:  $\text{toList } (\text{app } t1 \ t2) = (\text{toList } t1) \ @ \ (\text{toList } t2)$ 
  by (simp add: app3-list app-def)

```

1.2.7 Splitting

```

type-synonym ('e,'a) SplitDigit =
  ('e,'a) Node list  $\times$  ('e,'a) Node  $\times$  ('e,'a) Node list
type-synonym ('e,'a) SplitTree =
  ('e,'a) FingerTreeStruc  $\times$  ('e,'a) Node  $\times$  ('e,'a) FingerTreeStruc

```

Auxiliary functions to create a correct finger tree even if the left or right digit is empty

```

fun deepL :: ('e,'a::monoid-add) Node list  $\Rightarrow$  ('e,'a) FingerTreeStruc
   $\Rightarrow$  ('e,'a) Digit  $\Rightarrow$  ('e,'a) FingerTreeStruc where
  deepL [] m sf = (case (viewLn m) of None  $\Rightarrow$  digitToTree sf |
    (Some (a, m2))  $\Rightarrow$  deep (nodeToDigit a) m2 sf) |
  deepL pr m sf = deep (nlistToDigit pr) m sf
fun deepR :: ('e,'a::monoid-add) Digit  $\Rightarrow$  ('e,'a) FingerTreeStruc
   $\Rightarrow$  ('e,'a) Node list  $\Rightarrow$  ('e,'a) FingerTreeStruc where
  deepR pr m [] = (case (viewRn m) of None  $\Rightarrow$  digitToTree pr |
    (Some (a, m2))  $\Rightarrow$  deep pr m2 (nodeToDigit a)) |
  deepR pr m sf = deep pr m (nlistToDigit sf)

```

Splitting a list of nodes

```

fun splitNlist :: ('a::monoid-add  $\Rightarrow$  bool)  $\Rightarrow$  'a  $\Rightarrow$  ('e,'a) Node list
   $\Rightarrow$  ('e,'a) SplitDigit where
  splitNlist p i [a] = ([],a,[]) |
  splitNlist p i (a#b) =
    (let i2 = (i + gmn a) in
     (if (p i2)
      then ([],a,b)
      else
       (let (l,x,r) = (splitNlist p i2 b) in ((a#l),x,r))))

```

Splitting a digit by converting it into a list of nodes

```

definition splitDigit :: ('a::monoid-add  $\Rightarrow$  bool)  $\Rightarrow$  'a  $\Rightarrow$  ('e,'a) Digit
   $\Rightarrow$  ('e,'a) SplitDigit where
  splitDigit p i d = splitNlist p i (digitToNlist d)

```

Creating a finger tree from list of nodes

```

definition nlistToTree :: ('e,'a::monoid-add) Node list

```

$\Rightarrow ('e, 'a)$ *FingerTreeStruc* **where**
nlistToTree xs = lconsNlist xs Empty

Recursive splitting into a left and right tree and a center node

fun *nsplitTree* :: ('a::monoid-add \Rightarrow bool) \Rightarrow 'a \Rightarrow ('e, 'a) *FingerTreeStruc*
 \Rightarrow ('e, 'a) *SplitTree* **where**
nsplitTree p i Empty = (Empty, Tip undefined undefined, Empty)
 — Making the function total |
nsplitTree p i (Single ea) = (Empty, ea, Empty) |
nsplitTree p i (Deep - pr m sf) =
 (let
 vpr = (i + gmd pr);
 vm = (vpr + gmft m)
 in
 if (p vpr) then
 (let (l,x,r) = (*splitDigit p i pr*) in
 (*nlistToTree l,x,deepL r m sf*))
 else (if (p vm) then
 (let (ml,xs,mr) = (*nsplitTree p vpr m*);
 (l,x,r) = (*splitDigit p (vpr + gmft ml) (nodeToDigit xs)*) in
 (*deepR pr ml l,x,deepL r mr sf*))
 else
 (let (l,x,r) = (*splitDigit p vm sf*) in
 (*deepR pr m l,x,nlistToTree r*))
))

lemma *nlistToTree-inv*:

$\forall x \in \text{set } nl. \text{is-measured-node } x \implies \text{is-measured-ftree } (nlistToTree \text{ } nl)$
 $\forall x \in \text{set } nl. \text{is-leveln-node } n \ x \implies \text{is-leveln-ftree } n \ (nlistToTree \text{ } nl)$
by (*unfold nlistToTree-def, induct nl, auto simp add: nlcons-invmeas*)
 (*induct nl, auto simp add: nlcons-invlevel*)

lemma *nlistToTree-list*: *toList (nlistToTree nl) = nlistToList nl*

by (*auto simp add: nlistToTree-def nlistListLCons*)

lemma *deepL-inv*:

assumes *is-leveln-ftree (Suc n) m \wedge is-measured-ftree m*
and *is-leveln-digit n sf \wedge is-measured-digit sf*
and $\forall x \in \text{set } pr. (is-measured-node \ x \ \wedge \ is-leveln-node \ n \ x) \ \wedge \ \text{length } pr \leq 4$
shows *is-leveln-ftree n (deepL pr m sf) \wedge is-measured-ftree (deepL pr m sf)*
apply (*insert assms*)
apply (*induct pr m sf rule: deepL.induct*)
apply (*simp split: viewnres-split*)
apply *auto[1]*
apply (*simp-all add: digitToTree-inv deep-def*)

proof —

fix *m sf Node FingerTreeStruc*

assume *is-leveln-ftree (Suc n) m is-measured-ftree m*


```

      is-leveln-digit n sf is-measured-digit sf
      viewLn m = Some (Node, FingerTreeStruc)
thus is-leveln-digit n (nodeToDigit Node)
      ∧ is-leveln-ftree (Suc n) FingerTreeStruc
      by (simp add: viewLn-inv[of m Suc n Node FingerTreeStruc] nodeToDigit-inv)
next
fix m sf Node FingerTreeStruc
assume assms1:
  is-leveln-ftree (Suc n) m is-measured-ftree m
  is-leveln-digit n sf is-measured-digit sf
  viewLn m = Some (Node, FingerTreeStruc)
thus is-measured-digit (nodeToDigit Node) ∧ is-measured-ftree FingerTreeStruc
apply (auto simp only: viewLn-inv[of m Suc n Node FingerTreeStruc])
proof -
  from assms1 have is-measured-node Node ∧ is-leveln-node (Suc n) Node
    by (simp add: viewLn-inv[of m Suc n Node FingerTreeStruc])
  thus is-measured-digit (nodeToDigit Node)
    by (auto simp add: nodeToDigit-inv)
qed
next
fix v va
assume
  is-measured-node v ∧ is-leveln-node n (v::('a,'b) Node) ∧
  length (va::('a, 'b) Node list) ≤ 3 ∧
  (∀ x ∈ set va. is-measured-node x ∧ is-leveln-node n x ∧ length va ≤ 3)
thus is-leveln-digit n (nlistToDigit (v # va))
  ∧ is-measured-digit (nlistToDigit (v # va))
by(cases v#va rule: nlistToDigit.cases,simp-all)
qed

```

lemma *nlistToDigit-list*:

```

assumes 1 ≤ length xs ∧ length xs ≤ 4
shows digitToList(nlistToDigit xs) = nlistToList xs
by (insert assms, cases xs rule: nlistToDigit.cases,auto)

```

lemma *deepL-list*:

```

assumes is-leveln-ftree (Suc n) m ∧ is-measured-ftree m
and is-leveln-digit n sf ∧ is-measured-digit sf
and ∀ x ∈ set pr. (is-measured-node x ∧ is-leveln-node n x) ∧ length pr ≤ 4
shows toList (deepL pr m sf) = nlistToList pr @ toList m @ digitToList sf
proof (insert assms, induct pr m sf rule: deepL.induct)
case (1 m sf)
thus ?case
proof (auto split: viewnres-split simp add: deep-def)
  assume viewLn m = None
  hence m = Empty by (metis viewLn-empty)
  hence toList m = [] by simp

```

```

thus toList (digitToTree sf) = toList m @ digitToList sf
  by (simp add: digitToTree-list)
next
fix nd t
assume viewLn m = Some (nd, t)
  is-leveln-ftree (Suc n) m is-measured-ftree m
hence nodeToList nd @ toList t = toList m by (metis viewLn-list)
thus digitToList (nodeToDigit nd) @ toList t = toList m
  by (simp add: nodeToDigit-list)
qed
next
case (2 v va m sf)
thus ?case
  apply (unfold deepL.simps)
  apply (simp add: deep-def)
  apply (simp add: nlistToDigit-list)
done
qed

lemma deepR-inv:
assumes is-leveln-ftree (Suc n) m  $\wedge$  is-measured-ftree m
and is-leveln-digit n pr  $\wedge$  is-measured-digit pr
and  $\forall x \in \text{set } sf. (is-measured-node x \wedge is-leveln-node n x) \wedge \text{length } sf \leq 4$ 
shows is-leveln-ftree n (deepR pr m sf)  $\wedge$  is-measured-ftree (deepR pr m sf)
apply (insert assms)
apply (induct pr m sf rule: deepR.induct)
apply (simp split: viewnres-split)
apply auto[1]
apply (simp-all add: digitToTree-inv deep-def)
proof -
fix m pr Node FingerTreeStruc
assume is-leveln-ftree (Suc n) m is-measured-ftree m
  is-leveln-digit n pr is-measured-digit pr
  viewRn m = Some (Node, FingerTreeStruc)
thus
  is-leveln-digit n (nodeToDigit Node)
   $\wedge$  is-leveln-ftree (Suc n) FingerTreeStruc
  by (simp add: viewRn-inv[of m Suc n Node FingerTreeStruc] nodeToDigit-inv)
next
fix m pr Node FingerTreeStruc
assume assms1:
  is-leveln-ftree (Suc n) m is-measured-ftree m
  is-leveln-digit n pr is-measured-digit pr
  viewRn m = Some (Node, FingerTreeStruc)
thus is-measured-ftree FingerTreeStruc  $\wedge$  is-measured-digit (nodeToDigit Node)
  apply (auto simp only: viewRn-inv[of m Suc n Node FingerTreeStruc])
proof -
from assms1 have is-measured-node Node  $\wedge$  is-leveln-node (Suc n) Node
  by (simp add: viewRn-inv[of m Suc n Node FingerTreeStruc])

```

```

thus is-measured-digit (nodeToDigit Node)
  by (auto simp add: nodeToDigit-inv)
qed
next
fix v va
assume
  is-measured-node v  $\wedge$  is-leveln-node n (v:: ('a,'b) Node)  $\wedge$ 
  length (va:: ('a, 'b) Node list)  $\leq 3$   $\wedge$ 
  ( $\forall x \in \text{set } va. \text{is-measured-node } x \wedge \text{is-leveln-node } n \ x \wedge \text{length } va \leq 3$ )
thus is-leveln-digit n (nlistToDigit (v # va))  $\wedge$ 
  is-measured-digit (nlistToDigit (v # va))
  by(cases v#va rule: nlistToDigit.cases,simp-all)
qed

```

lemma *deepR-list*:

```

assumes is-leveln-ftree (Suc n) m  $\wedge$  is-measured-ftree m
and is-leveln-digit n pr  $\wedge$  is-measured-digit pr
and  $\forall x \in \text{set } sf. (\text{is-measured-node } x \wedge \text{is-leveln-node } n \ x) \wedge \text{length } sf \leq 4$ 
shows toList (deepR pr m sf) = digitToList pr @ toList m @ nlistToList sf
proof (insert assms, induct pr m sf rule: deepR.induct)
case (1 pr m)
thus ?case
proof (auto split: viewnres-split simp add: deep-def)
  assume viewRn m = None
  hence m = Empty by (metis viewRn-empty)
  hence toList m = [] by simp
  thus toList (digitToTree pr) = digitToList pr @ toList m
    by (simp add: digitToTree-list)
next
fix nd t
assume viewRn m = Some (nd, t) is-leveln-ftree (Suc n) m
  is-measured-ftree m
  hence toList t @ nodeToList nd = toList m by (metis viewRn-list)
  thus toList t @ digitToList (nodeToDigit nd) = toList m
    by (simp add: nodeToDigit-list)
qed
next
case (2 pr m v va)
thus ?case
  apply (unfold deepR.simps)
  apply (simp add: deep-def)
  apply (simp add: nlistToDigit-list)
done
qed

```

```

primrec gmnl:: ('e, 'a::monoid-add) Node list  $\Rightarrow$  'a where
  gmnl [] = 0 |
  gmnl (x#xs) = gmnl x + gmnl xs

```

lemma *gmnl-correct*:
assumes $\forall x \in \text{set } xs. \text{is-measured-node } x$
shows $\text{gmnl } xs = \text{sum-list } (\text{map } \text{snd } (\text{nlistToList } xs))$
by (*insert assms, induct xs*) (*auto simp add: add.assoc gmnl-correct*)

lemma *splitNlist-correct*: \llbracket
 $\bigwedge(a::'a) (b::'a). p \ a \implies p \ (a + b);$
 $\neg p \ i;$
 $p \ (i + \text{gmnl } (\text{nl} :: ('e, 'a)::\text{monoid-add}) \ \text{Node } \text{list});$
 $\text{splitNlist } p \ i \ \text{nl} = (l, n, r)$
 $\rrbracket \implies$
 $\neg p \ (i + (\text{gmnl } l))$
 \wedge
 $p \ (i + (\text{gmnl } l) + (\text{gmnl } n))$
 \wedge
 $\text{nl} = l @ n \# r$

proof (*induct p i nl arbitrary: l n r rule: splitNlist.induct*)
case 1 thus ?case by simp

next

case ($2 \ p \ i \ a \ v \ va \ l \ n \ r$) **note** $IV = \text{this}$

show ?case

proof (*cases p (i + (gmnl a))*)

case True with IV show ?thesis by simp

next

case False note IV2 = this IV thus ?thesis

proof –

obtain $l1 \ n1 \ r1$ **where**

$v1[\text{simp}]: \text{splitNlist } p \ (i + \text{gmnl } a) \ (v \# va) = (l1, n1, r1)$

by (*cases splitNlist p (i + gmnl a) (v # va), blast*)

note $miv = IV2(2)[\text{of } i + \text{gmnl } a \ l1 \ n1 \ r1]$

have $v2:p \ (i + \text{gmnl } a + \text{gmnl } (v \# va))$

using $IV2(5)$ **by** (*simp add: add.assoc*)

note $miv2 = miv[OF - IV2(1) IV2(3) IV2(1) \ v2 \ v1]$

have $v3: a \# l1 = l \ n1 = n \ r1 = r$ **using** $IV2 \ v1$ **by auto**

with $miv2$ **have**

$v4: \neg p \ (i + \text{gmnl } a + \text{gmnl } l1) \wedge$

$p \ (i + \text{gmnl } a + \text{gmnl } l1 + \text{gmnl } n1) \wedge$

$v \# va = l1 @ n1 \# r1$

by auto

with $v2 \ v3$ **show** ?thesis

by (*auto simp add: add.assoc*)

qed

qed

next

case 3 thus ?case by simp

qed

lemma *digitToNlist-inv*:

is-measured-digit $d \implies (\forall x \in \text{set } (\text{digitToNlist } d). \text{is-measured-node } x)$

is-leveln-digit $n d \implies (\forall x \in \text{set } (\text{digitToNlist } d). \text{is-leveln-node } n x)$

by (*cases* d , *auto*)(*cases* d , *auto*)

lemma *gmnl-gmd*:

is-measured-digit $d \implies \text{gmnl } (\text{digitToNlist } d) = \text{gmd } d$

by (*cases* d , *auto simp add: add.assoc*)

lemma *gmn-gmd*:

is-measured-node $nd \implies \text{gmd } (\text{nodeToDigit } nd) = \text{gmn } nd$

by (*auto simp add: nodeToDigit-inv nodeToDigit-list gmn-correct gmd-correct*)

lemma *splitDigit-inv*:

\llbracket
 $\bigwedge (a::'a) (b::'a). p a \implies p (a + b);$
 $\neg p i;$
is-measured-digit $d;$
is-leveln-digit $n d;$
 $p (i + \text{gmd } (d :: ('e, 'a)::\text{monoid-add } \text{Digit}));$
 $\text{splitDigit } p i d = (l, nd, r)$

$\rrbracket \implies$

$\neg p (i + (\text{gmnl } l))$

\wedge

$p (i + (\text{gmnl } l) + (\text{gmn } nd))$

\wedge

$(\forall x \in \text{set } l. (\text{is-measured-node } x \wedge \text{is-leveln-node } n x))$

\wedge

$(\forall x \in \text{set } r. (\text{is-measured-node } x \wedge \text{is-leveln-node } n x))$

\wedge

$(\text{is-measured-node } nd \wedge \text{is-leveln-node } n nd)$

proof –

fix $p i d n l nd r$

assume *assms*: $\bigwedge a b. p a \implies p (a + b) \neg p i$ *is-measured-digit* d

$p (i + \text{gmd } d)$ *splitDigit* $p i d = (l, nd, r)$

is-leveln-digit $n d$

from *assms*(3, 4) **have** $v1: p (i + \text{gmnl } (\text{digitToNlist } d))$

by (*simp add: gmnl-gmd*)

note *snc* = *splitNlist-correct* [of $p i$ *digitToNlist* $d l nd r$]

from *assms*(5) **have** $v2: \text{splitNlist } p i (\text{digitToNlist } d) = (l, nd, r)$

by (*simp add: splitDigit-def*)

note *snc1* = *snc*[OF *assms*(1) *assms*(2) $v1 v2$]

hence $v3: \neg p (i + \text{gmnl } l) \wedge p (i + \text{gmnl } l + \text{gmn } nd) \wedge$

$\text{digitToNlist } d = l @ nd \# r$ **by** *auto*

from *assms*(3,6) **have**

$v4: \forall x \in \text{set } (\text{digitToNlist } d). \text{is-measured-node } x$

$\forall x \in \text{set } (\text{digitToNlist } d). \text{is-leveln-node } n x$

by(*auto simp add: digitToNlist-inv*)

with $v3$ **have** $v5$: $\forall x \in \text{set } l. (\text{is-measured-node } x \wedge \text{is-leveln-node } n \ x)$
 $\forall x \in \text{set } r. (\text{is-measured-node } x \wedge \text{is-leveln-node } n \ x)$
 $\text{is-measured-node } nd \wedge \text{is-leveln-node } n \ nd$ **by** *auto*
with $v3$ $v5$ **show**
 $\neg p (i + \text{gmnl } l) \wedge p (i + \text{gmnl } l + \text{gmn } nd) \wedge$
 $(\forall x \in \text{set } l. \text{is-measured-node } x \wedge \text{is-leveln-node } n \ x) \wedge$
 $(\forall x \in \text{set } r. \text{is-measured-node } x \wedge \text{is-leveln-node } n \ x) \wedge$
 $\text{is-measured-node } nd \wedge \text{is-leveln-node } n \ nd$
by *auto*
qed

lemma *splitDigit-inv'*:

\llbracket
 $\text{splitDigit } p \ i \ d = (l, nd, r);$
 $\text{is-measured-digit } d;$
 $\text{is-leveln-digit } n \ d$
 $\rrbracket \implies$
 $(\forall x \in \text{set } l. (\text{is-measured-node } x \wedge \text{is-leveln-node } n \ x))$
 \wedge
 $(\forall x \in \text{set } r. (\text{is-measured-node } x \wedge \text{is-leveln-node } n \ x))$
 \wedge
 $(\text{is-measured-node } nd \wedge \text{is-leveln-node } n \ nd)$

apply (*unfold splitDigit-def*)
apply (*cases d*)
apply (*auto split: if-split-asm simp add: Let-def*)
done

lemma *splitDigit-list*: $\text{splitDigit } p \ i \ d = (l, n, r) \implies$
 $(\text{digitToList } d) = (\text{nlistToList } l) \ @ \ (\text{nodeToList } n) \ @ \ (\text{nlistToList } r)$
 $\wedge \text{length } l \leq 4 \wedge \text{length } r \leq 4$
apply (*unfold splitDigit-def*)
apply (*cases d*)
apply (*auto split: if-split-asm simp add: Let-def*)
done

lemma *gmnl-gmft*: $\forall x \in \text{set } nl. \text{is-measured-node } x \implies$
 $\text{gmft } (\text{nlistToTree } nl) = \text{gmnl } nl$
by (*auto simp add: gmnl-correct[of nl] nlistToTree-list[of nl]*
 $\text{nlistToTree-inv[of nl] gmft-correct[of nlistToTree nl]}$)

lemma *gmftR-gmnl*:

assumes $\text{is-leveln-ftree } (\text{Suc } n) \ m \wedge \text{is-measured-ftree } m$
and $\text{is-leveln-digit } n \ pr \wedge \text{is-measured-digit } pr$
and $\forall x \in \text{set } sf. (\text{is-measured-node } x \wedge \text{is-leveln-node } n \ x) \wedge \text{length } sf \leq 4$
shows $\text{gmft } (\text{deepR } pr \ m \ sf) = \text{gmd } pr + \text{gmft } m + \text{gmnl } sf$
proof –

```

from assms have
  v1: toList (deepR pr m sf) = digitToList pr @ toList m @ nlistToList sf
by (auto simp add: deepR-list)
from assms have
  v2: is-measured-ftree (deepR pr m sf)
by (auto simp add: deepR-inv)
with v1 have
  v3: gmft (deepR pr m sf) =
    sum-list (map snd (digitToList pr @ toList m @ nlistToList sf))
by (auto simp add: gmft-correct)
have
  v4: gmd pr + gmft m + gmnl sf =
    sum-list (map snd (digitToList pr @ toList m @ nlistToList sf))
by (auto simp add: gmd-correct gmft-correct gmnl-correct assms add.assoc)
with v3 show ?thesis by simp
qed

```

```

lemma nsplitTree-invpres: [
  is-leveln-ftree n (s:: ('e, 'a::monoid-add) FingerTreeStruc);
  is-measured-ftree s;
  ¬ p i;
  p (i + (gmft s));
  (nsplitTree p i s) = (l, nd, r)]
⇒
is-leveln-ftree n l
^
is-measured-ftree l
^
is-leveln-ftree n r
^
is-measured-ftree r
^
is-leveln-node n nd
^
is-measured-node nd

```

```

proof (induct p i s arbitrary: n l nd r rule: nsplitTree.induct)
  case 1
  thus ?case by auto
next
  case 2 thus ?case by auto
next
  case (∃ p i uu pr m sf n l nd r)
  thus ?case
  proof (cases p (i + gmd pr))
    case True with ∃ show ?thesis
    proof –
      obtain l1 x r1 where
        l1xr1: splitDigit p i pr = (l1, x, r1)

```

by (*cases splitDigit p i pr, blast*)
with *True 3* **have**
v1: l = nlistToTree l1 nd = x r = deepL r1 m sf **by** *auto*
from *l1x1* **have**
v2: digitToList pr = nlistToList l1 @ nodeToList x @ nlistToList r1
length l1 ≤ 4 length r1 ≤ 4
by (*auto simp add: splitDigit-list*)
from *3(2,3)* **have**
pr-m-sf-inv: is-leveln-digit n pr ∧ is-measured-digit pr
is-leveln-ftree (Suc n) m ∧ is-measured-ftree m
is-leveln-digit n sf ∧ is-measured-digit sf **by** *simp-all*
with *3(4,5) pr-m-sf-inv(1) True l1x1*
splitDigit-inv'[of p i pr l1 x r1 n] **have**
l1-x-r1-inv:
 $\forall x \in \text{set } l1. (\text{is-measured-node } x \wedge \text{is-leveln-node } n \ x)$
 $\forall x \in \text{set } r1. (\text{is-measured-node } x \wedge \text{is-leveln-node } n \ x)$
is-measured-node x ∧ is-leveln-node n x
by *auto*
from *l1-x-r1-inv v1 v2(3) pr-m-sf-inv* **have**
ziel3: is-leveln-ftree n l ∧ is-measured-ftree l ∧
is-leveln-ftree n r ∧ is-measured-ftree r ∧
is-leveln-node n nd ∧ is-measured-node nd
by (*auto simp add: nlistToTree-inv deepL-inv*)
thus *?thesis* **by** *simp*
qed
next
case *False* **note** *case1 = this with 3 show ?thesis*
proof (*cases p (i + gmd pr + gmft m)*)
case *False* **with** *case1 3* **show** *?thesis*
proof –
obtain *l1 x r1* **where**
l1x1: splitDigit p (i + gmd pr + gmft m) sf = (l1,x,r1)
by (*cases splitDigit p (i + gmd pr + gmft m) sf, blast*)
with *case1 False 3* **have**
v1: l = deepR pr m l1 nd = x r = nlistToTree r1 **by** *auto*
from *l1x1* **have**
v2: digitToList sf = nlistToList l1 @ nodeToList x @ nlistToList r1
length l1 ≤ 4 length r1 ≤ 4
by (*auto simp add: splitDigit-list*)
from *3(2,3)* **have**
pr-m-sf-inv: is-leveln-digit n pr ∧ is-measured-digit pr
is-leveln-ftree (Suc n) m ∧ is-measured-ftree m
is-leveln-digit n sf ∧ is-measured-digit sf **by** *simp-all*
from *3* **have**
v7: p (i + gmd pr + gmft m + gmd sf) **by** (*auto simp add: add.assoc*)
with *pr-m-sf-inv 3(4) pr-m-sf-inv(3) case1 False l1x1*
splitDigit-inv'[of p i + gmd pr + gmft m sf l1 x r1 n]
have *l1-x-r1-inv:*
 $\forall x \in \text{set } l1. (\text{is-measured-node } x \wedge \text{is-leveln-node } n \ x)$


```

     $\forall x \in \text{set } r1. (\text{is-measured-node } x \wedge \text{is-leveln-node } n \ x)$ 
     $\text{is-measured-node } x \wedge \text{is-leveln-node } n \ x$ 
    by auto
  from l1-x-r1-inv v1 v2(2) pr-m-sf-inv have
    ziel3:  $\text{is-leveln-ftree } n \ l \wedge \text{is-measured-ftree } l \wedge$ 
     $\text{is-leveln-ftree } n \ r \wedge \text{is-measured-ftree } r \wedge$ 
     $\text{is-leveln-node } n \ nd \wedge \text{is-measured-node } nd$ 
    by (auto simp add: nlistToTree-inv deepR-inv)
  from ziel3 show ?thesis by simp
qed
next
case True with case1 3 show ?thesis
proof -
  obtain l1 x r1 where
    l1-x-r1 :  $\text{nsplitTree } p \ (i + \text{gmd } pr) \ m = (l1, x, r1)$ 
    by (cases nsplitTree p (i + gmd pr) m, blast)
  from 3(2,3) have
    pr-m-sf-inv:  $\text{is-leveln-digit } n \ pr \wedge \text{is-measured-digit } pr$ 
     $\text{is-leveln-ftree } (\text{Suc } n) \ m \wedge \text{is-measured-ftree } m$ 
     $\text{is-leveln-digit } n \ sf \wedge \text{is-measured-digit } sf$  by simp-all
  with True case1
    3.hyps[of i + gmd pr i + gmd pr + gmft m Suc n l1 x r1]
    3(6) l1-x-r1
  have l1-x-r1-inv:
     $\text{is-leveln-ftree } (\text{Suc } n) \ l1 \wedge \text{is-measured-ftree } l1$ 
     $\text{is-leveln-ftree } (\text{Suc } n) \ r1 \wedge \text{is-measured-ftree } r1$ 
     $\text{is-leveln-node } (\text{Suc } n) \ x \wedge \text{is-measured-node } x$ 
    by auto
  obtain l2 x2 r2 where l2-x2-r2:
    splitDigit p (i + gmd pr + gmft l1) (nodeToDigit x) = (l2,x2,r2)
    by (cases splitDigit p (i + gmd pr + gmft l1) (nodeToDigit x),blast)
  from l1-x-r1-inv have
    ndx-inv:  $\text{is-leveln-digit } n \ (\text{nodeToDigit } x) \wedge$ 
     $\text{is-measured-digit } (\text{nodeToDigit } x)$ 
    by (auto simp add: nodeToDigit-inv gmn-gmd)
  note spdi = splitDigit-inv'[of p i + gmd pr + gmft l1
    nodeToDigit x l2 x2 r2 n]
  from ndx-inv l1-x-r1-inv(1) l2-x2-r2 3(4) have
    l2-x2-r2-inv:
     $\forall x \in \text{set } l2. \text{is-measured-node } x \wedge \text{is-leveln-node } n \ x$ 
     $\forall x \in \text{set } r2. \text{is-measured-node } x \wedge \text{is-leveln-node } n \ x$ 
     $\text{is-measured-node } x2 \wedge \text{is-leveln-node } n \ x2$ 
    by (auto simp add: spdi)
  note spdl = splitDigit-list[of p i + gmd pr + gmft l1
    nodeToDigit x l2 x2 r2]
  from l2-x2-r2 have
    l2-x2-r2-list:
    digitToList (nodeToDigit x) =
    nlistToList l2 @ nodeToList x2 @ nlistToList r2

```

```

    length l2 ≤ 4 ∧ length r2 ≤ 4
  by (auto simp add: spd1)
from case1 True 3(6) l1-x-r1 l2-x2-r2 have
  l-nd-r:
  l = deepR pr l1 l2
  nd = x2
  r = deepL r2 r1 sf
  by auto
note dr1 = deepR-inv[OF l1-x-r1-inv(1) pr-m-sf-inv(1)]
from dr1 l2-x2-r2-inv l2-x2-r2-list(2) l-nd-r have
  l-inv: is-leveln-ftree n l ∧ is-measured-ftree l
  by simp
note dl1 = deepL-inv[OF l1-x-r1-inv(2) pr-m-sf-inv(3)]
from dl1 l2-x2-r2-inv l2-x2-r2-list(2) l-nd-r have
  r-inv: is-leveln-ftree n r ∧ is-measured-ftree r
  by simp
from l2-x2-r2-inv l-nd-r have
  nd-inv: is-leveln-node n nd ∧ is-measured-node nd
  by simp
from l-inv r-inv nd-inv
show ?thesis by simp
qed
qed
qed
qed

```

lemma *nsplitTree-correct*: [

```

  is-leveln-ftree n (s:: ('e,'a::monoid-add) FingerTreeStruc);
  is-measured-ftree s;
  ∧(a::'a) (b::'a). p a ⇒ p (a + b);
  ¬ p i;
  p (i + (gmft s));
  (nsplitTree p i s) = (l, nd, r)]
⇒ (toList s) = (toList l) @ (nodeToList nd) @ (toList r)
∧
¬ p (i + (gmft l))
∧
p (i + (gmft l) + (gmn nd))
∧
is-leveln-ftree n l
∧
is-measured-ftree l
∧
is-leveln-ftree n r
∧
is-measured-ftree r
∧
is-leveln-node n nd
∧

```

is-measured-node nd

```

proof (induct p i s arbitrary: n l nd r rule: nsplitTree.induct)
  case 1
  thus ?case by auto
next
  case 2 thus ?case by auto
next
  case (3 p i uu pr m sf n l nd r)
  thus ?case
  proof (cases p (i + gmd pr))
    case True with 3 show ?thesis
    proof -
      obtain l1 x r1 where
        l1xr1: splitDigit p i pr = (l1,x,r1)
        by (cases splitDigit p i pr, blast)
      with True 3(7) have
        v1: l = nlistToTree l1 nd = x r = deepL r1 m sf by auto
      from l1xr1 have
        v2: digitToList pr = nlistToList l1 @ nodeToList x @ nlistToList r1
        length l1 ≤ 4 length r1 ≤ 4
        by (auto simp add: splitDigit-list)
      from 3(2,3) have
        pr-m-sf-inv: is-leveln-digit n pr ∧ is-measured-digit pr
        is-leveln-ftree (Suc n) m ∧ is-measured-ftree m
        is-leveln-digit n sf ∧ is-measured-digit sf by simp-all
      with 3(4,5) pr-m-sf-inv(1) True l1xr1
        splitDigit-inv[of p i pr n l1 x r1] have
        l1-x-r1-inv:
          ¬ p (i + (gmn l1))
          p (i + (gmn l1) + (gmn x))
          ∀ x ∈ set l1. (is-measured-node x ∧ is-leveln-node n x)
          ∀ x ∈ set r1. (is-measured-node x ∧ is-leveln-node n x)
          is-measured-node x ∧ is-leveln-node n x
        by auto
      from v2 v1 l1-x-r1-inv(4) pr-m-sf-inv have
        ziel1: toList (Deep uu pr m sf) = toList l @ nodeToList nd @ toList r
        by (auto simp add: nlistToTree-list deepL-list)
      from l1-x-r1-inv(3) v1(1) have
        v3: gmft l = gmn l1 by (simp add: gmn-gmft)
      with l1-x-r1-inv(1,2) v1 have
        ziel2: ¬ p (i + gmft l)
        p (i + gmft l + gmn nd)
        by simp-all
      from l1-x-r1-inv(3,4,5) v1 v2(3) pr-m-sf-inv have
        ziel3: is-leveln-ftree n l ∧ is-measured-ftree l ∧
        is-leveln-ftree n r ∧ is-measured-ftree r ∧
        is-leveln-node n nd ∧ is-measured-node nd
        by (auto simp add: nlistToTree-inv deepL-inv)

```

```

from ziel1 ziel2 ziel3 show ?thesis by simp
qed
next
case False note case1 = this with 3 show ?thesis
proof (cases p (i + gmd pr + gmft m))
case False with case1 3 show ?thesis
proof –
obtain l1 x r1 where
  l1xr1: splitDigit p (i + gmd pr + gmft m) sf = (l1,x,r1)
  by (cases splitDigit p (i + gmd pr + gmft m) sf, blast)
with case1 False 3(7) have
  v1: l = deepR pr m l1 nd = x r = nlistToTree r1 by auto
from l1xr1 have
  v2: digitToList sf = nlistToList l1 @ nodeToList x @ nlistToList r1
  length l1 ≤ 4 length r1 ≤ 4
  by (auto simp add: splitDigit-list)
from 3(2,3) have
  pr-m-sf-inv: is-leveln-digit n pr ∧ is-measured-digit pr
  is-leveln-ftree (Suc n) m ∧ is-measured-ftree m
  is-leveln-digit n sf ∧ is-measured-digit sf by simp-all
from 3(3,6) have
  v7: p (i + gmd pr + gmft m + gmd sf) by (auto simp add: add.assoc)
with pr-m-sf-inv 3(4) pr-m-sf-inv(3) case1 False l1xr1
  splitDigit-inv[of p i + gmd pr + gmft m sf n l1 x r1]
have l1-x-r1-inv:
   $\neg p (i + gmd pr + gmft m + gmn l1)$ 
   $p (i + gmd pr + gmft m + gmn l1 + gmn x)$ 
   $\forall x \in \text{set } l1. (\text{is-measured-node } x \wedge \text{is-leveln-node } n x)$ 
   $\forall x \in \text{set } r1. (\text{is-measured-node } x \wedge \text{is-leveln-node } n x)$ 
   $\text{is-measured-node } x \wedge \text{is-leveln-node } n x$ 
  by auto
from v2 v1 l1-x-r1-inv(3) pr-m-sf-inv have
  ziel1: toList (Deep uu pr m sf) = toList l @ nodeToList nd @ toList r
  by (auto simp add: nlistToTree-list deepR-list)
from l1-x-r1-inv(4) v1(3) have
  v3: gmft r = gmn l r1 by (simp add: gmn-gmft)
with l1-x-r1-inv(1,2,3) pr-m-sf-inv v1 v2 have
  ziel2:  $\neg p (i + gmft l)$ 
   $p (i + gmft l + gmn nd)$ 
  by (auto simp add: gmftR-gmn add.assoc)
from l1-x-r1-inv(3,4,5) v1 v2(2) pr-m-sf-inv have
  ziel3: is-leveln-ftree n l ∧ is-measured-ftree l ∧
  is-leveln-ftree n r ∧ is-measured-ftree r ∧
  is-leveln-node n nd ∧ is-measured-node nd
  by (auto simp add: nlistToTree-inv deepR-inv)
from ziel1 ziel2 ziel3 show ?thesis by simp
qed
next
case True with case1 3 show ?thesis

```

proof –

obtain $l1\ x\ r1$ **where**

$l1\text{-}x\text{-}r1 : nsplitTree\ p\ (i + gmd\ pr)\ m = (l1,\ x,\ r1)$

by (cases $nsplitTree\ p\ (i + gmd\ pr)\ m$) *blast*

from $\exists(2,3)$ **have**

$pr\text{-}m\text{-}sf\text{-}inv : is\text{-}leveln\text{-}digit\ n\ pr \wedge is\text{-}measured\text{-}digit\ pr$

$is\text{-}leveln\text{-}ftree\ (Suc\ n)\ m \wedge is\text{-}measured\text{-}ftree\ m$

$is\text{-}leveln\text{-}digit\ n\ sf \wedge is\text{-}measured\text{-}digit\ sf$ **by** *simp-all*

with *True case1*

$\exists.hyps[of\ i + gmd\ pr\ i + gmd\ pr + gmft\ m\ Suc\ n\ l1\ x\ r1]$

$\exists(4)\ l1\text{-}x\text{-}r1$

have $l1\text{-}x\text{-}r1\text{-}inv :$

$\neg\ p\ (i + gmd\ pr + gmft\ l1)$

$p\ (i + gmd\ pr + gmft\ l1 + gmn\ x)$

$is\text{-}leveln\text{-}ftree\ (Suc\ n)\ l1 \wedge is\text{-}measured\text{-}ftree\ l1$

$is\text{-}leveln\text{-}ftree\ (Suc\ n)\ r1 \wedge is\text{-}measured\text{-}ftree\ r1$

$is\text{-}leveln\text{-}node\ (Suc\ n)\ x \wedge is\text{-}measured\text{-}node\ x$

and $l1\text{-}x\text{-}r1\text{-}list :$

$toList\ m = toList\ l1\ @\ nodeToList\ x\ @\ toList\ r1$

by *auto*

obtain $l2\ x2\ r2$ **where** $l2\text{-}x2\text{-}r2 :$

$splitDigit\ p\ (i + gmd\ pr + gmft\ l1)\ (nodeToDigit\ x) = (l2,\ x2,\ r2)$

by (cases $splitDigit\ p\ (i + gmd\ pr + gmft\ l1)\ (nodeToDigit\ x)$, *blast*)

from $l1\text{-}x\text{-}r1\text{-}inv(2,5)$ **have**

$ndx\text{-}inv : is\text{-}leveln\text{-}digit\ n\ (nodeToDigit\ x) \wedge$

$is\text{-}measured\text{-}digit\ (nodeToDigit\ x)$

$p\ (i + gmd\ pr + gmft\ l1 + gmd\ (nodeToDigit\ x))$

by (*auto simp add: nodeToDigit-inv gmn-gmd*)

note $spdi = splitDigit\text{-}inv[of\ p\ i + gmd\ pr + gmft\ l1$

$nodeToDigit\ x\ n\ l2\ x2\ r2]$

from $ndx\text{-}inv\ l1\text{-}x\text{-}r1\text{-}inv(1)\ l2\text{-}x2\text{-}r2\ \exists(4)$ **have**

$l2\text{-}x2\text{-}r2\text{-}inv : \neg\ p\ (i + gmd\ pr + gmft\ l1 + gmn\ l2)$

$p\ (i + gmd\ pr + gmft\ l1 + gmn\ l2 + gmn\ x2)$

$\forall x \in set\ l2. is\text{-}measured\text{-}node\ x \wedge is\text{-}leveln\text{-}node\ n\ x$

$\forall x \in set\ r2. is\text{-}measured\text{-}node\ x \wedge is\text{-}leveln\text{-}node\ n\ x$

$is\text{-}measured\text{-}node\ x2 \wedge is\text{-}leveln\text{-}node\ n\ x2$

by (*auto simp add: spdi*)

note $spdl = splitDigit\text{-}list[of\ p\ i + gmd\ pr + gmft\ l1$

$nodeToDigit\ x\ l2\ x2\ r2]$

from $l2\text{-}x2\text{-}r2$ **have**

$l2\text{-}x2\text{-}r2\text{-}list :$

$digitToList\ (nodeToDigit\ x) =$

$nlistToList\ l2\ @\ nodeToList\ x2\ @\ nlistToList\ r2$

$length\ l2 \leq 4 \wedge length\ r2 \leq 4$

by (*auto simp add: spdl*)

from *case1 True* $\exists(7)\ l1\text{-}x\text{-}r1\ l2\text{-}x2\text{-}r2$ **have**

$l\text{-}nd\text{-}r :$

$l = deepR\ pr\ l1\ l2$

$nd = x2$

```

    r = deepL r2 r1 sf
  by auto
note dr1 = deepR-inv[OF l1-x-r1-inv(3) pr-m-sf-inv(1)]
from dr1 l2-x2-r2-inv(3) l2-x2-r2-list(2) l-nd-r have
  l-inv: is-leveln-ftree n l ∧ is-measured-ftree l
  by simp
note dl1 = deepL-inv[OF l1-x-r1-inv(4) pr-m-sf-inv(3)]
from dl1 l2-x2-r2-inv(4) l2-x2-r2-list(2) l-nd-r have
  r-inv: is-leveln-ftree n r ∧ is-measured-ftree r
  by simp
from l2-x2-r2-inv l-nd-r have
  nd-inv: is-leveln-node n nd ∧ is-measured-node nd
  by simp
from l-nd-r(1,2) l2-x2-r2-inv(1,2,3)
  l1-x-r1-inv(3) l2-x2-r2-list(2) pr-m-sf-inv(1)
have split-point:
  ¬ p (i + gmft l)
  p (i + gmft l + gmn nd)
  by (auto simp add: gmftR-gmnl add.assoc)
from l2-x2-r2-list have x-list:
  nodeList x = nlistToList l2 @ nodeList x2 @ nlistToList r2
  by (simp add: nodeListDigit-list)
from l1-x-r1-inv(3) pr-m-sf-inv(1)
  l2-x2-r2-inv(3) l2-x2-r2-list(2) l-nd-r(1)
have l-list: toList l = digitToList pr @ toList l1 @ nlistToList l2
  by (auto simp add: deepR-list)
from l1-x-r1-inv(4) pr-m-sf-inv(3) l2-x2-r2-inv(4)
  l2-x2-r2-list(2) l-nd-r(3)
have r-list: toList r = nlistToList r2 @ toList r1 @ digitToList sf
  by (auto simp add: deepL-list)
from x-list l1-x-r1-list l-list r-list l-nd-r
have toList (Deep uu pr m sf) = toList l @ nodeList nd @ toList r
  by auto
with split-point l-inv r-inv nd-inv
show ?thesis by simp
qed
qed
qed
qed

```

A predicate on the elements of a monoid is called *monotone*, iff, when it holds for some value a , it also holds for all values $a + b$:

Split a finger tree by a monotone predicate on the annotations, using a given initial value. Intuitively, the elements are summed up from left to right, and the split is done when the predicate first holds for the sum. The predicate must not hold for the initial value of the summation, and must hold for the sum of all elements.

definition *splitTree*

$:: ('a::\text{monoid-add} \Rightarrow \text{bool}) \Rightarrow 'a \Rightarrow ('e, 'a) \text{FingerTreeStruc}$
 $\Rightarrow ('e, 'a) \text{FingerTreeStruc} \times ('e \times 'a) \times ('e, 'a) \text{FingerTreeStruc}$
where
 $\text{splitTree } p \ i \ t = (\text{let } (l, x, r) = \text{nsplitTree } p \ i \ t \ \text{in } (l, (n\text{-unwrap } x), r))$

lemma *splitTree-invpres*:

assumes *inv*: *ft-invar* (*s*: ('e,'a)::monoid-add) *FingerTreeStruc*)

assumes *init-ff*: $\neg p \ i$

assumes *sum-tt*: $p \ (i + \text{annot } s)$

assumes *fmt*: $(\text{splitTree } p \ i \ s) = (l, (e, a), r)$

shows *ft-invar* *l* **and** *ft-invar* *r*

proof –

obtain *l1 nd r1* **where**

l1-nd-r1: $\text{nsplitTree } p \ i \ s = (l1, nd, r1)$

by (*cases nsplitTree p i s, blast*)

with *assms* **have**

l0: $l = l1$

$(e, a) = n\text{-unwrap } nd$

$r = r1$

by (*auto simp add: splitTree-def*)

note *nsp* = *nsplitTree-invpres*[*of 0 s p i l1 nd r1*]

from *assms* **have** $p \ (i + \text{gmft } s)$ **by** (*simp add: ft-invar-def annot-def*)

with *assms l1-nd-r1 l0* **have**

v1:

is-leveln-ftree $0 \ l \wedge \text{is-measured-ftree } l$

is-leveln-ftree $0 \ r \wedge \text{is-measured-ftree } r$

is-leveln-node $0 \ nd \wedge \text{is-measured-node } nd$

by (*auto simp add: nsp ft-invar-def*)

thus *ft-invar* *l* **and** *ft-invar* *r*

by (*simp-all add: ft-invar-def annot-def*)

qed

lemma *splitTree-correct*:

assumes *inv*: *ft-invar* (*s*: ('e,'a)::monoid-add) *FingerTreeStruc*)

assumes *mono*: $\forall a \ b. p \ a \longrightarrow p \ (a + b)$

assumes *init-ff*: $\neg p \ i$

assumes *sum-tt*: $p \ (i + \text{annot } s)$

assumes *fmt*: $(\text{splitTree } p \ i \ s) = (l, (e, a), r)$

shows $(\text{toList } s) = (\text{toList } l) \ @ \ (e, a) \ \# \ (\text{toList } r)$

and $\neg p \ (i + \text{annot } l)$

and $p \ (i + \text{annot } l + a)$

and *ft-invar* *l* **and** *ft-invar* *r*

proof –

obtain *l1 nd r1* **where**

l1-nd-r1: $\text{nsplitTree } p \ i \ s = (l1, nd, r1)$

by (*cases nsplitTree p i s, blast*)

with *assms* **have**
 $l0: l = l1$
 $(e,a) = n\text{-unwrap } nd$
 $r = r1$
by (*auto simp add: splitTree-def*)
note $nsp = nsplitTree\text{-correct}[of\ 0\ s\ p\ i\ l1\ nd\ r1]$

from *assms* **have** $p\ (i + gmft\ s)$ **by** (*simp add: ft-invar-def annot-def*)
with *assms l1-nd-r1 l0* **have**
 $v1:$
 $(toList\ s) = (toList\ l) \textcircled{\#} (nodeToList\ nd) \textcircled{\#} (toList\ r)$
 $\neg p\ (i + (gmft\ l))$
 $p\ (i + (gmft\ l) + (gmn\ nd))$
 $is\ leveln\ \text{ftree}\ 0\ l \wedge is\ measured\ \text{ftree}\ l$
 $is\ leveln\ \text{ftree}\ 0\ r \wedge is\ measured\ \text{ftree}\ r$
 $is\ leveln\ \text{node}\ 0\ nd \wedge is\ measured\ \text{node}\ nd$
by (*auto simp add: nsp ft-invar-def*)
from $v1(6)\ l0(2)$ **have**
 $ndea: nd = Tip\ e\ a$
by (*cases nd*) *auto*
hence $nd\ \text{list}\ \text{inv}: nodeToList\ nd = [(e,a)]$
 $gmn\ nd = a$ **by** *simp-all*
with $v1$ **show** $(toList\ s) = (toList\ l) \textcircled{\#} (e,a) \# (toList\ r)$
and $\neg p\ (i + annot\ l)$
and $p\ (i + annot\ l + a)$
and $ft\ \text{invar}\ l$ **and** $ft\ \text{invar}\ r$
by (*simp-all add: ft-invar-def annot-def*)

qed

lemma *splitTree-correctE*:
assumes $inv: ft\ \text{invar}\ (s: ('e,'a::monoid\ \text{add})\ \text{FingerTreeStruc})$
assumes $mono: \forall a\ b. p\ a \longrightarrow p\ (a + b)$
assumes $init\ \text{ff}: \neg p\ i$
assumes $sum\ \text{tt}: p\ (i + annot\ s)$
obtains $l\ e\ a\ r$ **where**
 $(splitTree\ p\ i\ s) = (l, (e,a), r)$ **and**
 $(toList\ s) = (toList\ l) \textcircled{\#} (e,a) \# (toList\ r)$ **and**
 $\neg p\ (i + annot\ l)$ **and**
 $p\ (i + annot\ l + a)$ **and**
 $ft\ \text{invar}\ l$ **and** $ft\ \text{invar}\ r$

proof –
obtain $l\ e\ a\ r$ **where** $fmt: (splitTree\ p\ i\ s) = (l, (e,a), r)$
by (*cases (splitTree p i s)*) *auto*
from *splitTree-correct*[*of s p, OF assms fmt*] *fmt*
show *?thesis*
by (*blast intro: that*)

qed

1.2.8 Folding

fun *foldl-node* :: ('s ⇒ 'e × 'a ⇒ 's) ⇒ 's ⇒ ('e,'a) Node ⇒ 's **where**
foldl-node f σ (Tip e a) = f σ (e,a)|
foldl-node f σ (Node2 - a b) = *foldl-node* f (foldl-node f σ a) b|
foldl-node f σ (Node3 - a b c) =
foldl-node f (foldl-node f (foldl-node f σ a) b) c

primrec *foldl-digit* :: ('s ⇒ 'e × 'a ⇒ 's) ⇒ 's ⇒ ('e,'a) Digit ⇒ 's **where**
foldl-digit f σ (One n1) = *foldl-node* f σ n1|
foldl-digit f σ (Two n1 n2) = *foldl-node* f (foldl-node f σ n1) n2|
foldl-digit f σ (Three n1 n2 n3) =
foldl-node f (foldl-node f (foldl-node f σ n1) n2) n3|
foldl-digit f σ (Four n1 n2 n3 n4) =
foldl-node f (foldl-node f (foldl-node f (foldl-node f σ n1) n2) n3) n4

primrec *foldr-node* :: ('e × 'a ⇒ 's ⇒ 's) ⇒ ('e,'a) Node ⇒ 's ⇒ 's **where**
foldr-node f (Tip e a) σ = f (e,a) σ |
foldr-node f (Node2 - a b) σ = *foldr-node* f a (foldr-node f b σ)|
foldr-node f (Node3 - a b c) σ
= *foldr-node* f a (foldr-node f b (foldr-node f c σ))

primrec *foldr-digit* :: ('e × 'a ⇒ 's ⇒ 's) ⇒ ('e,'a) Digit ⇒ 's ⇒ 's **where**
foldr-digit f (One n1) σ = *foldr-node* f n1 σ|
foldr-digit f (Two n1 n2) σ = *foldr-node* f n1 (foldr-node f n2 σ)|
foldr-digit f (Three n1 n2 n3) σ =
foldr-node f n1 (foldr-node f n2 (foldr-node f n3 σ))|
foldr-digit f (Four n1 n2 n3 n4) σ =
foldr-node f n1 (foldr-node f n2 (foldr-node f n3 (foldr-node f n4 σ)))

lemma *foldl-node-correct*:
foldl-node f σ nd = List.foldl f σ (nodeToList nd)
by (induct nd arbitrary: σ) (auto simp add: nodeToList-def)

lemma *foldl-digit-correct*:
foldl-digit f σ d = List.foldl f σ (digitToList d)
by (induct d arbitrary: σ) (auto
simp add: digitToList-def foldl-node-correct)

lemma *foldr-node-correct*:
foldr-node f nd σ = List.foldr f (nodeToList nd) σ
by (induct nd arbitrary: σ) (auto simp add: nodeToList-def)

lemma *foldr-digit-correct*:
foldr-digit f d σ = List.foldr f (digitToList d) σ
by (induct d arbitrary: σ) (auto
simp add: digitToList-def foldr-node-correct)

Fold from left

primrec *foldl* :: ('s ⇒ 'e × 'a ⇒ 's) ⇒ 's ⇒ ('e,'a) *FingerTreeStruc* ⇒ 's
where
foldl *f* *σ* *Empty* = *σ* |
foldl *f* *σ* (*Single* *nd*) = *foldl-node* *f* *σ* *nd* |
foldl *f* *σ* (*Deep* - *d1* *m* *d2*) =
foldl-digit *f* (*foldl* *f* (*foldl-digit* *f* *σ* *d1*) *m*) *d2*

lemma *foldl-correct*:
foldl *f* *σ* *t* = *List.foldl* *f* *σ* (*toList* *t*)
by (*induct* *t* *arbitrary*: *σ*) (*auto*
simp *add*: *toList-def* *foldl-node-correct* *foldl-digit-correct*)

Fold from right

primrec *foldr* :: ('e × 'a ⇒ 's ⇒ 's) ⇒ ('e,'a) *FingerTreeStruc* ⇒ 's ⇒ 's
where
foldr *f* *Empty* *σ* = *σ* |
foldr *f* (*Single* *nd*) *σ* = *foldr-node* *f* *nd* *σ* |
foldr *f* (*Deep* - *d1* *m* *d2*) *σ*
= *foldr-digit* *f* *d1* (*foldr* *f* *m* (*foldr-digit* *f* *d2* *σ*))

lemma *foldr-correct*:
foldr *f* *t* *σ* = *List.foldr* *f* (*toList* *t*) *σ*
by (*induct* *t* *arbitrary*: *σ*) (*auto*
simp *add*: *toList-def* *foldr-node-correct* *foldr-digit-correct*)

1.2.9 Number of elements

primrec *count-node* :: ('e, 'a) *Node* ⇒ *nat* **where**
count-node (*Tip* - *a*) = 1 |
count-node (*Node2* - *a* *b*) = *count-node* *a* + *count-node* *b* |
count-node (*Node3* - *a* *b* *c*) = *count-node* *a* + *count-node* *b* + *count-node* *c*

primrec *count-digit* :: ('e,'a) *Digit* ⇒ *nat* **where**
count-digit (*One* *a*) = *count-node* *a* |
count-digit (*Two* *a* *b*) = *count-node* *a* + *count-node* *b* |
count-digit (*Three* *a* *b* *c*) = *count-node* *a* + *count-node* *b* + *count-node* *c* |
count-digit (*Four* *a* *b* *c* *d*)
= *count-node* *a* + *count-node* *b* + *count-node* *c* + *count-node* *d*

lemma *count-node-correct*:
count-node *n* = *length* (*nodeToList* *n*)
by (*induct* *n*, *auto* *simp* *add*: *nodeToList-def* *count-node-def*)

lemma *count-digit-correct*:
count-digit *d* = *length* (*digitToList* *d*)
by (*cases* *d*, *auto* *simp* *add*: *digitToList-def* *count-digit-def* *count-node-correct*)

primrec *count* :: ('e,'a) *FingerTreeStruc* ⇒ *nat* **where**
count *Empty* = 0 |

```

count (Single a) = count-node a |
count (Deep - pr m sf) = count-digit pr + count m + count-digit sf

```

```

lemma count-correct[simp]:
  count t = length (toList t)
  by (induct t,
    auto simp add: toList-def count-def
      count-digit-correct count-node-correct)
end

```

interpretation *FingerTreeStruc*: *FingerTreeStruc-loc* .

```

no-notation FingerTreeStruc.lcons (infixr < 65)
no-notation FingerTreeStruc.rcons (infixl > 65)

```

1.3 Hiding the invariant

In this section, we define the datatype of all FingerTrees that fulfill their invariant, and define the operations to work on this datatype. The advantage is, that the correctness lemmas do no longer contain explicit invariant predicates, what makes them more handy to use.

1.3.1 Datatype

```

typedef (overloaded) ('e, 'a) FingerTree =
  { t :: ('e, 'a)::monoid-add) FingerTreeStruc. FingerTreeStruc.ft-invar t }
proof -
  have Empty ∈ ?FingerTree by (simp)
  then show ?thesis ..
qed

```

```

lemma Rep-FingerTree-invar[simp]: FingerTreeStruc.ft-invar (Rep-FingerTree t)
  using Rep-FingerTree by simp

```

```

lemma [simp]:
  FingerTreeStruc.ft-invar t ⇒ Rep-FingerTree (Abs-FingerTree t) = t
  using Abs-FingerTree-inverse by simp

```

```

lemma [simp, code abstype]: Abs-FingerTree (Rep-FingerTree t) = t
  by (rule Rep-FingerTree-inverse)

```

```

typedef (overloaded) ('e, 'a) viewres =
  { r :: (('e × 'a) × ('e, 'a)::monoid-add) FingerTreeStruc) option .
  case r of None ⇒ True | Some (a, t) ⇒ FingerTreeStruc.ft-invar t }
apply (rule-tac x=None in exI)
apply auto
done

```

lemma [*simp*, *code abstype*]: *Abs-viewres* (*Rep-viewres* x) = x
by (*rule Rep-viewres-inverse*)

lemma *Abs-viewres-inverse-None*[*simp*]:
Rep-viewres (*Abs-viewres* *None*) = *None*
by (*simp add: Abs-viewres-inverse*)

lemma *Abs-viewres-inverse-Some*:
FingerTreeStruc.ft-invar $t \implies$
Rep-viewres (*Abs-viewres* (*Some* (a,t))) = *Some* (a,t)
by (*auto simp add: Abs-viewres-inverse*)

definition [*code*]: *extract-viewres-isNone* $r ==$ *Rep-viewres* $r =$ *None*

definition [*code*]: *extract-viewres-a* $r ==$
case (*Rep-viewres* r) *of* *Some* (a,t) \Rightarrow a

definition *extract-viewres-t* $r ==$
case (*Rep-viewres* r) *of* *None* \Rightarrow *Abs-FingerTree* *Empty*
| *Some* (a,t) \Rightarrow *Abs-FingerTree* t

lemma [*code abstract*]: *Rep-FingerTree* (*extract-viewres-t* r) =
(*case* (*Rep-viewres* r) *of* *None* \Rightarrow *Empty* | *Some* (a,t) \Rightarrow t)

apply (*cases* r)

apply (*auto split: option.split option.split-asm*)

simp add: extract-viewres-t-def Abs-viewres-inverse-Some)

done

definition *extract-viewres* $r ==$

if *extract-viewres-isNone* r *then* *None*

else *Some* (*extract-viewres-a* r , *extract-viewres-t* r)

typedef (**overloaded**) ($'e,'a$) *splitres* =

{ ($(l,a,r)::('e,'a)$ *FingerTreeStruc* \times ($'e \times 'a$) \times ($'e,'a::$ *monoid-add*) *FingerTreeStruc*)

| l a r .

FingerTreeStruc.ft-invar $l \wedge$ *FingerTreeStruc.ft-invar* r }

apply (*rule-tac* $x=($ *Empty,undefined,Empty*) **in** *exI*)

apply *auto*

done

lemma [*simp*, *code abstype*]: *Abs-splitres* (*Rep-splitres* x) = x
by (*rule Rep-splitres-inverse*)

lemma *Abs-splitres-inverse*:

FingerTreeStruc.ft-invar $r \implies$ *FingerTreeStruc.ft-invar* $s \implies$

Rep-splitres (*Abs-splitres* ((r,a,s))) = (r,a,s)

by (*auto simp add: Abs-splitres-inverse*)

definition [*code*]: *extract-splitres-a* $r ==$ *case* (*Rep-splitres* r) *of* (l,a,s) \Rightarrow a

definition *extract-splitres-l* $r ==$ *case* (*Rep-splitres* r) *of* (l,a,r) \Rightarrow

Abs-FingerTree l

lemma [code abstract]: *Rep-FingerTree* (extract-splitres-l r) = (case (Rep-splitres r) of (l,a,r) ⇒ l)
apply (cases r)
apply (auto split: option.split option.split-asm
simp add: extract-splitres-l-def Abs-splitres-inverse)
done
definition extract-splitres-r r == case (Rep-splitres r) of (l,a,r) ⇒
Abs-FingerTree r
lemma [code abstract]: *Rep-FingerTree* (extract-splitres-r r) = (case (Rep-splitres r) of (l,a,r) ⇒ r)
apply (cases r)
apply (auto split: option.split option.split-asm
simp add: extract-splitres-r-def Abs-splitres-inverse)
done

definition extract-splitres r ==
(extract-splitres-l r,
extract-splitres-a r,
extract-splitres-r r)

1.3.2 Definition of Operations

locale *FingerTree-loc*

begin

definition [code]: toList t == *FingerTreeStruc.toList* (*Rep-FingerTree* t)
definition empty where empty == *Abs-FingerTree FingerTreeStruc.Empty*
lemma [code abstract]: *Rep-FingerTree* empty = *FingerTreeStruc.Empty*
by (simp add: empty-def)

lemma empty-rep: t=empty ⇔ *Rep-FingerTree* t = *Empty*
apply (auto simp add: empty-def)
apply (metis *Rep-FingerTree-inverse*)
done

definition [code]: annot t == *FingerTreeStruc.annot* (*Rep-FingerTree* t)

definition toTree t == *Abs-FingerTree* (*FingerTreeStruc.toTree* t)

lemma [code abstract]: *Rep-FingerTree* (toTree t) = *FingerTreeStruc.toTree* t
by (simp add: toTree-def)

definition lcons a t ==

Abs-FingerTree (*FingerTreeStruc.lcons* a (*Rep-FingerTree* t))

lemma [code abstract]:

Rep-FingerTree (lcons a t) = (*FingerTreeStruc.lcons* a (*Rep-FingerTree* t))

by (simp add: lcons-def *FingerTreeStruc.lcons-correct*)

definition rcons t a ==

Abs-FingerTree (*FingerTreeStruc.rcons* (*Rep-FingerTree* t) a)

lemma [code abstract]:

Rep-FingerTree (rcons t a) = (*FingerTreeStruc.rcons* (*Rep-FingerTree* t) a)

by (simp add: rcons-def *FingerTreeStruc.rcons-correct*)

definition *viewL-aux* $t ==$
Abs-viewres (*FingerTreeStruc.viewL* (*Rep-FingerTree* t))
definition *viewL* $t ==$ *extract-viewres* (*viewL-aux* t)
lemma [*code abstract*]:
Rep-viewres (*viewL-aux* t) = (*FingerTreeStruc.viewL* (*Rep-FingerTree* t))
apply (*cases* (*FingerTreeStruc.viewL* (*Rep-FingerTree* t)))
apply (*auto simp add: viewL-aux-def*)
apply (*cases* *Rep-FingerTree* $t =$ *Empty*)
apply *simp*
apply (*auto*
elim!: *FingerTreeStruc.viewL-correct-nonEmpty*
[*of Rep-FingerTree* t , *simplified*]
simp add: Abs-viewres-inverse-Some)
done

definition *viewR-aux* $t ==$
Abs-viewres (*FingerTreeStruc.viewR* (*Rep-FingerTree* t))
definition *viewR* $t ==$ *extract-viewres* (*viewR-aux* t)
lemma [*code abstract*]:
Rep-viewres (*viewR-aux* t) = (*FingerTreeStruc.viewR* (*Rep-FingerTree* t))
apply (*cases* (*FingerTreeStruc.viewR* (*Rep-FingerTree* t)))
apply (*auto simp add: viewR-aux-def*)
apply (*cases* *Rep-FingerTree* $t =$ *Empty*)
apply *simp*
apply (*auto*
elim!: *FingerTreeStruc.viewR-correct-nonEmpty*
[*of Rep-FingerTree* t , *simplified*]
simp add: Abs-viewres-inverse-Some)
done

definition [*code*]: *isEmpty* $t ==$ *FingerTreeStruc.isEmpty* (*Rep-FingerTree* t)

definition [*code*]: *head* $t =$ *FingerTreeStruc.head* (*Rep-FingerTree* t)

definition *tail* $t \equiv$

if $t =$ *empty* *then*

empty

else

Abs-FingerTree (*FingerTreeStruc.tail* (*Rep-FingerTree* t))

— Make function total, to allow abstraction

lemma [*code abstract*]: *Rep-FingerTree* (*tail* t) =

(*if* (*FingerTreeStruc.isEmpty* (*Rep-FingerTree* t)) *then* *Empty*

else *FingerTreeStruc.tail* (*Rep-FingerTree* t))

apply (*simp add: tail-def FingerTreeStruc.tail-correct FingerTreeStruc.isEmpty-def empty-rep*)

apply (*auto simp add: empty-def*)

done

definition [*code*]: *headR* $t =$ *FingerTreeStruc.headR* (*Rep-FingerTree* t)

definition *tailR* $t \equiv$

```

    if t=empty then
      empty
    else
      Abs-FingerTree (FingerTreeStruc.tailR (Rep-FingerTree t))
lemma [code abstract]: Rep-FingerTree (tailR t) =
  (if (FingerTreeStruc.isEmpty (Rep-FingerTree t)) then Empty
   else FingerTreeStruc.tailR (Rep-FingerTree t))
apply (simp add: tailR-def FingerTreeStruc.tailR-correct FingerTreeStruc.isEmpty-def
empty-rep)
apply (simp add: empty-def)
done

```

```

definition app s t = Abs-FingerTree (
  FingerTreeStruc.app (Rep-FingerTree s) (Rep-FingerTree t))

```

```

lemma [code abstract]:
  Rep-FingerTree (app s t) =
  FingerTreeStruc.app (Rep-FingerTree s) (Rep-FingerTree t)
by (simp add: app-def FingerTreeStruc.app-correct)

```

```

definition splitTree-aux p i t == if (¬p i ∧ p (i+annot t)) then
  Abs-splitres (FingerTreeStruc.splitTree p i (Rep-FingerTree t))
else
  Abs-splitres (Empty,undefined,Empty)

```

```

definition splitTree p i t == extract-splitres (splitTree-aux p i t)

```

```

lemma [code abstract]:
  Rep-splitres (splitTree-aux p i t) = (if (¬p i ∧ p (i+annot t)) then
  (FingerTreeStruc.splitTree p i (Rep-FingerTree t))
else
  (Empty,undefined,Empty))
using FingerTreeStruc.splitTree-Invpres[of Rep-FingerTree t p i]
apply (auto simp add: splitTree-aux-def annot-def Abs-splitres-inverse)
apply (cases FingerTreeStruc.splitTree p i (Rep-FingerTree t))
apply (force simp add: Abs-FingerTree-inverse Abs-splitres-inverse)
done

```

```

definition foldl where

```

```

  [code]: foldl f σ t == FingerTreeStruc.foldl f σ (Rep-FingerTree t)

```

```

definition foldr where

```

```

  [code]: foldr f t σ == FingerTreeStruc.foldr f (Rep-FingerTree t) σ

```

```

definition count where

```

```

  [code]: count t == FingerTreeStruc.count (Rep-FingerTree t)

```

1.3.3 Correctness statements

```

lemma empty-correct: toList t = [] ↔ t=empty
apply (unfold toList-def empty-rep)
apply (simp add: FingerTreeStruc.toList-empty)
done

```

```

lemma toList-of-empty[simp]: toList empty = []
  apply (unfold toList-def empty-def)
  apply (auto simp add: FingerTreeStruc.toList-empty)
  done

lemma annot-correct: annot t = sum-list (map snd (toList t))
  apply (unfold toList-def annot-def)
  apply (simp add: FingerTreeStruc.annot-correct)
  done

lemma toTree-correct: toList (toTree l) = l
  apply (unfold toList-def toTree-def)
  apply (simp add: FingerTreeStruc.toTree-correct)
  done

lemma lcons-correct: toList (lcons a t) = a#toList t
  apply (unfold toList-def lcons-def)
  apply (simp add: FingerTreeStruc.lcons-correct)
  done

lemma rcons-correct: toList (rcons t a) = toList t@[a]
  apply (unfold toList-def rcons-def)
  apply (simp add: FingerTreeStruc.rcons-correct)
  done

lemma viewL-correct:
  t = empty  $\implies$  viewL t = None
  t  $\neq$  empty  $\implies$   $\exists a s. \text{viewL } t = \text{Some } (a,s) \wedge \text{toList } t = a\#\text{toList } s$ 
  apply (unfold toList-def viewL-def viewL-aux-def
    extract-viewres-def extract-viewres-isNone-def
    extract-viewres-a-def
    extract-viewres-t-def
    empty-rep)
  apply (simp add: FingerTreeStruc.viewL-correct)
  apply (drule FingerTreeStruc.viewL-correct(2)[OF Rep-FingerTree-invar])
  apply (auto simp add: Abs-viewres-inverse)
  done

lemma viewL-empty[simp]: viewL empty = None
  using viewL-correct by auto

lemma viewL-nonEmpty:
  assumes t $\neq$ empty
  obtains a s where viewL t = Some (a,s) toList t = a#toList s
  using assms viewL-correct by blast

lemma viewR-correct:
  t = empty  $\implies$  viewR t = None

```



```

t ≠ empty ⇒ ∃ a s. viewR t = Some (a,s) ∧ toList t = toList s@[a]
apply (unfold toList-def viewR-def viewR-aux-def
  extract-viewres-def extract-viewres-isNone-def
  extract-viewres-a-def
  extract-viewres-t-def
  empty-rep)
apply (simp add: FingerTreeStruc.viewR-correct)
apply (drule FingerTreeStruc.viewR-correct(2)[OF Rep-FingerTree-invar])
apply (auto simp add: Abs-viewres-inverse)
done

```

```

lemma viewR-empty[simp]: viewR empty = None
using viewR-correct by auto

```

```

lemma viewR-nonEmpty:
assumes t ≠ empty
obtains a s where viewR t = Some (a,s) toList t = toList s@[a]
using assms viewR-correct by blast

```

```

lemma isEmpty-correct: isEmpty t ⇔ t = empty
apply (unfold toList-def isEmpty-def empty-rep)
apply (simp add: FingerTreeStruc.isEmpty-correct FingerTreeStruc.toList-empty)
done

```

```

lemma head-correct: t ≠ empty ⇒ head t = hd (toList t)
apply (unfold toList-def head-def empty-rep)
apply (simp add: FingerTreeStruc.head-correct)
done

```

```

lemma tail-correct: t ≠ empty ⇒ toList (tail t) = tl (toList t)
apply (unfold toList-def tail-def empty-rep)
apply (simp add: FingerTreeStruc.tail-correct)
done

```

```

lemma headR-correct: t ≠ empty ⇒ headR t = last (toList t)
apply (unfold toList-def headR-def empty-rep)
apply (simp add: FingerTreeStruc.headR-correct)
done

```

```

lemma tailR-correct: t ≠ empty ⇒ toList (tailR t) = butlast (toList t)
apply (unfold toList-def tailR-def empty-rep)
apply (simp add: FingerTreeStruc.tailR-correct)
done

```

```

lemma app-correct: toList (app s t) = toList s @ toList t
apply (unfold toList-def app-def)
apply (simp add: FingerTreeStruc.app-correct)
done

```

```

lemma splitTree-correct:
  assumes mono:  $\forall a b. p a \longrightarrow p (a + b)$ 
  assumes init-ff:  $\neg p i$ 
  assumes sum-tt:  $p (i + \text{annot } s)$ 
  assumes fmt:  $(\text{splitTree } p \ i \ s) = (l, (e,a), r)$ 
  shows  $(\text{toList } s) = (\text{toList } l) \ @ \ (e,a) \ \# \ (\text{toList } r)$ 
  and  $\neg p (i + \text{annot } l)$ 
  and  $p (i + \text{annot } l + a)$ 
  apply (rule
    FingerTreeStruc.splitTree-correctE[
      where  $p=p$  and  $s=\text{Rep-FingerTree } s$ ,
      OF - mono init-ff sum-tt[unfolded annot-def],
      simplified
    ])
  using fmt
  apply (unfold toList-def splitTree-aux-def splitTree-def annot-def
    extract-splitres-def extract-splitres-l-def
    extract-splitres-a-def extract-splitres-r-def) [1]
  apply (auto split: if-split-asm prod.split-asm
    simp add: init-ff sum-tt[unfolded annot-def] Abs-splitres-inverse) [1]

apply (rule
  FingerTreeStruc.splitTree-correctE[
    where  $p=p$  and  $s=\text{Rep-FingerTree } s$ ,
    OF - mono init-ff sum-tt[unfolded annot-def],
    simplified
  ])
using fmt
apply (unfold toList-def splitTree-aux-def splitTree-def annot-def
  extract-splitres-def extract-splitres-l-def
  extract-splitres-a-def extract-splitres-r-def) [1]
apply (auto split: if-split-asm prod.split-asm
  simp add: init-ff sum-tt[unfolded annot-def] Abs-splitres-inverse) [1]

apply (rule
  FingerTreeStruc.splitTree-correctE[
    where  $p=p$  and  $s=\text{Rep-FingerTree } s$ ,
    OF - mono init-ff sum-tt[unfolded annot-def],
    simplified
  ])
using fmt
apply (unfold toList-def splitTree-aux-def splitTree-def annot-def
  extract-splitres-def extract-splitres-l-def
  extract-splitres-a-def extract-splitres-r-def) [1]
apply (auto split: if-split-asm prod.split-asm
  simp add: init-ff sum-tt[unfolded annot-def] Abs-splitres-inverse) [1]
done

```

```

lemma splitTree-correctE:

```

```

assumes mono:  $\forall a b. p a \longrightarrow p (a + b)$ 
assumes init-ff:  $\neg p i$ 
assumes sum-tt:  $p (i + \text{annot } s)$ 
obtains l e a r where
  (splitTree p i s) = (l, (e,a), r) and
  (toList s) = (toList l) @ (e,a) # (toList r) and
   $\neg p (i + \text{annot } l)$  and
   $p (i + \text{annot } l + a)$ 
proof –
  obtain l e a r where fmt: (splitTree p i s) = (l, (e,a), r)
    by (cases (splitTree p i s)) auto
  from splitTree-correct[of p, OF assms fmt] fmt
  show ?thesis
    by (blast intro: that)
qed

```

```

lemma foldl-correct:  $\text{foldl } f \sigma t = \text{List.foldl } f \sigma (\text{toList } t)$ 
  apply (unfold toList-def foldl-def)
  apply (simp add: FingerTreeStruc.foldl-correct)
  done

```

```

lemma foldr-correct:  $\text{foldr } f t \sigma = \text{List.foldr } f (\text{toList } t) \sigma$ 
  apply (unfold toList-def foldr-def)
  apply (simp add: FingerTreeStruc.foldr-correct)
  done

```

```

lemma count-correct:  $\text{count } t = \text{length } (\text{toList } t)$ 
  apply (unfold toList-def count-def)
  apply (simp add: FingerTreeStruc.count-correct)
  done

```

end

interpretation *FingerTree*: *FingerTree-loc* .

1.4 Interface Documentation

In this section, we list all supported operations on finger trees, along with a short plaintext documentation and their correctness statements.

$\frac{\textit{FingerTree.toList}::('a, 'b) \textit{FingerTree} \Rightarrow ('a \times 'b) \textit{list}}{\text{Convert to list } (O(n))}$

$\frac{\textit{FingerTree.empty}::('a, 'b) \textit{FingerTree}}{\text{The empty finger tree } (O(1))}$

Spec $\textit{FingerTree.empty-correct}$:

$(\textit{FingerTree.toList} \ ?t = []) = (?t = \textit{FingerTree.empty})$

$\frac{\textit{FingerTree.annot}::('a, 'b) \textit{FingerTree} \Rightarrow 'b}{\text{Return sum of all annotations } (O(1))}$

Spec $\textit{FingerTree.annot-correct}$:

$\textit{FingerTree.annot} \ ?t = \textit{sum-list} (\textit{map snd} (\textit{FingerTree.toList} \ ?t))$

$\frac{\textit{FingerTree.toTree}::('a \times 'b) \textit{list} \Rightarrow ('a, 'b) \textit{FingerTree}}{\text{Convert list to finger tree } (O(n \log(n)))}$

Spec $\textit{FingerTree.toTree-correct}$:

$\textit{FingerTree.toList} (\textit{FingerTree.toTree} \ ?l) = ?l$

$\frac{\textit{FingerTree.lcons}::'a \times 'b \Rightarrow ('a, 'b) \textit{FingerTree} \Rightarrow ('a, 'b) \textit{FingerTree}}{\text{Append element at the left end } (O(\log(n)), O(1) \text{ amortized})}$

Spec $\textit{FingerTree.lcons-correct}$:

$\textit{FingerTree.toList} (\textit{FingerTree.lcons} \ ?a \ ?t) = ?a \ \# \ \textit{FingerTree.toList} \ ?t$

$\frac{\textit{FingerTree.rcons}::('a, 'b) \textit{FingerTree} \Rightarrow 'a \times 'b \Rightarrow ('a, 'b) \textit{FingerTree}}{\text{Append element at the right end } (O(\log(n)), O(1) \text{ amortized})}$

Spec $\textit{FingerTree.rcons-correct}$:

$\textit{FingerTree.toList} (\textit{FingerTree.rcons} \ ?t \ ?a) = \textit{FingerTree.toList} \ ?t \ @ \ [?a]$

$\frac{\textit{FingerTree.viewL}::('a, 'b) \textit{FingerTree} \Rightarrow (('a \times 'b) \times ('a, 'b) \textit{FingerTree}) \textit{option}}{\text{Detach leftmost element } (O(\log(n)), O(1) \text{ amortized})}$

Spec $\textit{FingerTree.viewL-correct}$:

$?t = \textit{FingerTree.empty} \implies \textit{FingerTree.viewL} \ ?t = \textit{None}$

$?t \neq \textit{FingerTree.empty} \implies$

$\exists a \ s. \ \textit{FingerTree.viewL} \ ?t = \textit{Some} \ (a, \ s) \ \wedge$

$\textit{FingerTree.toList} \ ?t = a \ \# \ \textit{FingerTree.toList} \ s$

$\frac{\textit{FingerTree.viewR}::('a, 'b) \textit{FingerTree} \Rightarrow (('a \times 'b) \times ('a, 'b) \textit{FingerTree}) \textit{option}}{\text{Detach rightmost element } (O(\log(n)), O(1) \text{ amortized})}$

Spec $\textit{FingerTree.viewR-correct}$:

$?t = \text{FingerTree.empty} \implies \text{FingerTree.viewR } ?t = \text{None}$
 $?t \neq \text{FingerTree.empty} \implies$
 $\exists a \ s. \text{FingerTree.viewR } ?t = \text{Some } (a, s) \wedge$
 $\text{FingerTree.toList } ?t = \text{FingerTree.toList } s @ [a]$

$\text{FingerTree.isEmpty}::('a, 'b) \text{FingerTree} \Rightarrow \text{bool}$
 Check whether tree is empty ($O(1)$)
Spec *FingerTree.isEmpty-correct*:

$\text{FingerTree.isEmpty } ?t = (?t = \text{FingerTree.empty})$

$\text{FingerTree.head}::('a, 'b) \text{FingerTree} \Rightarrow 'a \times 'b$
 Get leftmost element of non-empty tree ($O(\log(n))$)
Spec *FingerTree.head-correct*:

$?t \neq \text{FingerTree.empty} \implies \text{FingerTree.head } ?t = \text{hd } (\text{FingerTree.toList } ?t)$

$\text{FingerTree.tail}::('a, 'b) \text{FingerTree} \Rightarrow ('a, 'b) \text{FingerTree}$
 Get all but leftmost element of non-empty tree ($O(\log(n))$)
Spec *FingerTree.tail-correct*:

$?t \neq \text{FingerTree.empty} \implies$
 $\text{FingerTree.toList } (\text{FingerTree.tail } ?t) = \text{tl } (\text{FingerTree.toList } ?t)$

$\text{FingerTree.headR}::('a, 'b) \text{FingerTree} \Rightarrow 'a \times 'b$
 Get rightmost element of non-empty tree ($O(\log(n))$)
Spec *FingerTree.headR-correct*:

$?t \neq \text{FingerTree.empty} \implies \text{FingerTree.headR } ?t = \text{last } (\text{FingerTree.toList } ?t)$

$\text{FingerTree.tailR}::('a, 'b) \text{FingerTree} \Rightarrow ('a, 'b) \text{FingerTree}$
 Get all but rightmost element of non-empty tree ($O(\log(n))$)
Spec *FingerTree.tailR-correct*:

$?t \neq \text{FingerTree.empty} \implies$
 $\text{FingerTree.toList } (\text{FingerTree.tailR } ?t) = \text{butlast } (\text{FingerTree.toList } ?t)$

$\text{FingerTree.app}::('a, 'b) \text{FingerTree} \Rightarrow ('a, 'b) \text{FingerTree} \Rightarrow ('a, 'b) \text{FingerTree}$
 Concatenate two finger trees ($O(\log(m+n))$)
Spec *FingerTree.app-correct*:

$\text{FingerTree.toList } (\text{FingerTree.app } ?s ?t) =$
 $\text{FingerTree.toList } ?s @ \text{FingerTree.toList } ?t$

FingerTree.splitTree

$\text{FingerTree.splitTree}::('a \Rightarrow \text{bool})$
 $\Rightarrow 'a \Rightarrow ('b, 'a) \text{FingerTree}$
 $\Rightarrow ('b, 'a) \text{FingerTree} \times$
 $('b \times 'a) \times ('b, 'a) \text{FingerTree}$

Split tree by a monotone predicate. ($O(\log(n))$)

A predicate p over the annotations is called monotone, iff, for all annotations a, b with $p(a)$, we have already $p(a + b)$.

Splitting is done by specifying a monotone predicate p that does not hold for the initial value i of the summation, but holds for i plus the sum of all annotations. The tree is then split at the position where p starts to hold for the sum of all elements up to that position.

Spec *FingerTree.splitTree-correct*:

$$\begin{aligned} & \llbracket \forall a b. ?p a \longrightarrow ?p (a + b); \neg ?p ?i; ?p (?i + \text{FingerTree.annot } ?s); \\ & \quad \text{FingerTree.splitTree } ?p ?i ?s = (?l, (?e, ?a), ?r) \rrbracket \\ \implies & \text{FingerTree.toList } ?s = \\ & \quad \text{FingerTree.toList } ?l @ (?e, ?a) \# \text{FingerTree.toList } ?r \\ & \llbracket \forall a b. ?p a \longrightarrow ?p (a + b); \neg ?p ?i; ?p (?i + \text{FingerTree.annot } ?s); \\ & \quad \text{FingerTree.splitTree } ?p ?i ?s = (?l, (?e, ?a), ?r) \rrbracket \\ \implies & \neg ?p (?i + \text{FingerTree.annot } ?l) \\ & \llbracket \forall a b. ?p a \longrightarrow ?p (a + b); \neg ?p ?i; ?p (?i + \text{FingerTree.annot } ?s); \\ & \quad \text{FingerTree.splitTree } ?p ?i ?s = (?l, (?e, ?a), ?r) \rrbracket \\ \implies & ?p (?i + \text{FingerTree.annot } ?l + ?a) \end{aligned}$$

FingerTree.foldl

FingerTree.foldl::('a ⇒ 'b × 'c ⇒ 'a) ⇒ 'a ⇒ ('b, 'c) FingerTree ⇒ 'a

Fold with function from left

Spec *FingerTree.foldl-correct*:

FingerTree.foldl ?f ?σ ?t = foldl ?f ?σ (FingerTree.toList ?t)

FingerTree.foldr

FingerTree.foldr::('a × 'b ⇒ 'c ⇒ 'c) ⇒ ('a, 'b) FingerTree ⇒ 'c ⇒ 'c

Fold with function from right

Spec *FingerTree.foldr-correct*:

FingerTree.foldr ?f ?t ?σ = foldr ?f (FingerTree.toList ?t) ?σ

FingerTree.count::('a, 'b) FingerTree ⇒ nat

Return the number of elements

Spec *FingerTree.count-correct*:

FingerTree.count ?t = length (FingerTree.toList ?t)

end

2 Related work

Finger trees were originally introduced by Hinze and Paterson[1], who give an implementation in Haskell. Our implementation closely follows this original implementation.

There is also a machine-checked formalization of 2-3 finger trees in Coq [2]. Like ours, it closely follows the original paper of Hinze and Paterson. The main difference is that the Coq-formalization encodes the invariants directly into the datatype for finger trees, while we first define the bigger algebraic datatype *FingerTreeStruc* along with the predicate *ft-invar* that checks the invariant. This bigger type and the *ft-invar*-predicate is then wrapped into the datatype *FingerTree*, that, however, exposes no algebraic structure any more. Our approach greatly simplifies matters in the context of Isabelle/HOL, as it can be realized with Isabelle’s datatype-package.

References

- [1] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, 2006.
- [2] M. Sozeau. Program-ing finger trees in coq. In *ICFP '07*, pages 13–24, New York, NY, USA, 2007. ACM.